OI 知识点复习笔记

dtcxzyw

May 20, 2019

前言

本人写复习笔记的目的有~~两个~~五个:

- 系统地复习知识点并挖掘一些有用的性质。
- 学会使用 LaTeX。
- 预先踩坑并记录, 而不是在考场上首次掉坑, 整场都在 Debug。
- 以此纪念我短暂的 OI 生涯, 并为将来的 ACM 做准备。
- ~~给学弟留下一些有用的东西~~。

当前共585页, 507,176(133,009) 字。
项目地址:https://github.com/dtcxzyw/OI-Source

声明

- **禁止将其用于商业目的。**
- **如需转载请标明出处,包括文末引用的链接/书籍/论文**
- **本人不承担因文中内容错误而导致损失的责任**

欢迎学弟(似乎并没有)与其他 OIer 指出文中错误或贡献更多新内容。

本笔记在准确性、普及度、分类等方面自然比不上 24OI 的 OI-Wiki [1]。毕竟到目前为止这份笔记只有我一人维护, 而 OI-Wiki 却有上百活跃贡献者。现阶段本笔记仅限于自用, 因此我没有介绍简单的算法 /数据结构。就算将来有学弟要用我也不会再补上这些内容, 我希望他们在读这份笔记之前已经有能力获得 NOIP 提高组一等奖。

[1] Wiki for OI / ICPC. (某大型游戏线上攻略, 内含炫酷算术魔法) https://oi-wiki.org/ https://github.com/24OI/OI-wiki

快速跳转

前言 ii

目录 v

1 博弈论 1

2 网络流 12

3 数据结构 60

4 数论 131

5 集合论 群论 190

6 组合数学 200

7 多项式 214

8 线性代数 253

9 数学杂项 271

10 动态规划 294

11 树 339

12 最短路相关问题 377

13 生成树 391

14 图论 409

15 字符串 440

16 计算几何 479

17 最优化问题 512

18 理论 528

19 杂项 532

20 总结 **564**

A 资料推荐 **569**

B Index **572**

C Bibliography **576**

后记 **577**

目录

前言 ii

目录 v

1 博弈论 1
1.1 SG 函数与 SG 定理 . . . 1
1.1.1 适用范围 . . . 1
1.1.2 SG 函数 . . . 2
1.1.3 SG 定理 . . . 3
1.2 Nim 系列游戏 . . . 5
1.2.1 Nim 游戏 . . . 5
1.2.2 Bash 游戏 . . . 5
1.2.3 NimK 游戏 . . . 5
1.2.4 Anti Nim . . . 6
1.2.5 阶梯博弈 . . . 6
1.2.6 威佐夫博弈 . . . 7
1.2.7 斐波那契博弈 . . . 8
1.3 Alpha-Beta 剪枝 . . . 9
1.3.1 Min-Max 搜索 . . . 9
1.3.2 Alpha-Beta 剪枝 . . . 11
1.4 本章注记 . . . 11

2 网络流 12
2.1 二分图 . . . 13
2.1.1 二分图判定 . . . 13
2.1.2 二分图最大匹配 . . . 14
2.1.3 二分图最大权匹配 Kuhn-Munkras Algorithm . . . 16
2.1.4 二分图常见模型 . . . 17
2.1.5 Hall 定理 . . . 18
2.2 最大流 . . . 20
2.2.1 Dinic 算法 . . . 20
2.2.2 ISAP 算法 . . . 29
2.2.3 HLPP 算法 . . . 31
2.2.4 最大流与最小割 . . . 38
2.2.5 无向图最小割 . . . 38
2.2.6 最小割性质 . . . 42
2.3 费用流 . . . 42
2.3.1 使用 Dijkstra 实现费用流 . . . 42

2.3.2 多路增广费用流 . . . 43
2.3.3 消圈定理 . . . 45
2.3.4 zkw 费用流 . . . 45
2.4 带上下界网络流 . . . 47
2.4.1 无源汇有上下界可行流 . . . 47
2.4.2 有上下界带源汇点可行流 . . . 47
2.4.3 有上下界带一组无限流量源汇点可行流 . . . 48
2.5 常见网络流/最小割模型 . . . 48
2.5.1 平面图转对偶图 . . . 48
2.5.2 最大权闭合子图 . . . 50
2.5.3 区间 k 覆盖问题 . . . 53
2.5.4 集合划分问题 . . . 53
2.6 最小割树 . . . 54
2.6.1 Gusfield 算法构造 . . . 54
2.6.2 询问 . . . 54
2.7 技巧总结 . . . 58
2.7.1 最大流 . . . 58
2.7.2 最小割 . . . 58
2.7.3 费用流 . . . 58

3 数据结构 60
3.1 树状数组 . . . 61
3.1.1 标号管辖范围 . . . 61
3.1.2 线性预处理 . . . 62
3.1.3 lowbit 函数原理 . . . 62
3.1.4 区间加法区间求和 . . . 62
3.2 线段树 . . . 62
3.2.1 技巧 . . . 62
3.2.2 zkw 线段树 . . . 63
3.2.3 势能分析线段树 . . . 63
3.2.4 Segment Tree Beats(吉司机线段树)与历史最值问题 . . . 63
3.2.5 线段树分治 . . . 69
3.2.6 线段树优化建图 . . . 73
3.2.7 猫树 . . . 76
3.2.8 线段树合并 . . . 83
3.2.9 线段树分裂 . . . 83
3.2.10 李超线段树 . . . 85
3.2.11 线段树维护单调栈 . . . 88
3.2.12 时间线段树 . . . 90
3.3 划分树 . . . 90
3.3.1 构建 . . . 91
3.3.2 查询 . . . 91
3.3.3 板子 . . . 91
3.4 二叉搜索树 . . . 93
3.4.1 二叉搜索树性质 . . . 93
3.4.2 FHQTreap . . . 93
3.4.3 splay . . . 97
3.5 动态树 . . . 99
3.5.1 常规操作 . . . 99
3.5.2 技巧/常见方法 . . . 102

3.6 并查集 . . . 111
3.6.1 路径压缩 . . . 111
3.6.2 按秩合并 . . . 111
3.6.3 复杂度证明 . . . 111
3.6.4 并查集的分裂 . . . 111
3.6.5 并查集重构树 . . . 113
3.7 K-D Tree . . . 115
3.7.1 构树 . . . 115
3.7.2 插入 . . . 116
3.7.3 删除 . . . 116
3.7.4 查询 . . . 116
3.7.5 估值 . . . 117
3.7.6 技巧 . . . 117
3.8 堆 . . . 118
3.8.1 左偏树 . . . 118
3.8.2 斜堆 . . . 119
3.8.3 可删堆 . . . 119
3.8.4 配对堆 . . . 120
3.9 可持久化数据结构 . . . 121
3.9.1 主席树 . . . 121
3.9.2 可持久化 Trie . . . 121
3.9.3 可持久化数组 . . . 121
3.9.4 优化 . . . 121
3.10 DLX 舞蹈链 . . . 124
3.10.1 X 算法 . . . 124
3.10.2 DLX . . . 125
3.11 Hash 表 . . . 128
3.11.1 链接法 . . . 128
3.11.2 双重散列 + 开放寻址法 . . . 128
3.11.3 完全散列 . . . 128
3.12 珂朵莉树/老司机树 . . . 128
3.12.1 存储 . . . 129
3.12.2 切分 . . . 129
3.12.3 区间赋值 . . . 129
3.12.4 应用 . . . 130
3.13 ETT . . . 130
3.13.1 欧拉遍历序列 . . . 130
3.13.2 基本操作 . . . 130

4 数论 131
4.1 辗转相除法 GCD . . . 132
4.1.1 裴蜀定理 . . . 132
4.1.2 exgcd . . . 133
4.1.3 位运算 gcd . . . 133
4.1.4 O(n)-O(1)gcd . . . 134
4.2 欧拉定理 . . . 137
4.2.1 费马小定理 . . . 137
4.2.2 线性推逆元 . . . 137
4.2.3 欧拉定理 . . . 138
4.2.4 扩展欧拉定理 . . . 138

4.3 素性测试 . . . 139
4.3.1 Miller Rabin 随机性素性测试 . . . 139
4.3.2 Baillie–PSW 素性测试 . . . 140
4.4 Pollard Rho 启发式因子分解 . . . 146
4.4.1 利用 Birthday Trick 提高效率 . . . 146
4.4.2 Floyd 判圈法 . . . 147
4.5 RSA 算法 . . . 147
4.5.1 原理 . . . 147
4.5.2 应用 . . . 148
4.5.3 RSA 破解 . . . 149
4.6 中国剩余定理 CRT . . . 149
4.6.1 CRT . . . 149
4.6.2 ExCRT . . . 149
4.7 积性函数与线性筛 . . . 150
4.7.1 定义 . . . 150
4.7.2 常见积性函数 . . . 150
4.7.3 线性筛 . . . 151
4.7.4 积性函数筛 . . . 152
4.7.5 因子分解 . . . 153
4.8 狄利克雷卷积, 狄利克雷逆与莫比乌斯反演 . . . 153
4.8.1 狄利克雷卷积 . . . 153
4.8.2 狄利克雷逆 . . . 154
4.8.3 莫比乌斯反演 . . . 154
4.8.4 常见技巧 . . . 155
4.9 低于线性时间复杂度筛法 . . . 157
4.9.1 杜教筛 . . . 157
4.9.2 min_25 筛 . . . 160
4.9.3 Powerful Number . . . 164
4.9.4 素数 k 次幂前缀和 . . . 166
4.9.5 约数个数函数前缀和 . . . 166
4.10 离散对数问题 . . . 166
4.10.1 BSGS . . . 166
4.10.2 单有原根模数多询问离散对数问题 . . . 169
4.10.3 Pollard rho 算法 . . . 169
4.10.4 Pohlig–Hellman 算法 . . . 172
4.10.5 对 1 求离散对数 . . . 176
4.11 原根 . . . 176
4.11.1 基本定义与定理 . . . 176
4.11.2 求模 n 的原根 . . . 176
4.11.3 原根的应用 . . . 177
4.12 二次剩余及其扩展 . . . 180
4.12.1 勒让德符号 . . . 180
4.12.2 二次剩余 . . . 181
4.12.3 三次剩余 . . . 182
4.12.4 高次剩余 . . . 183
4.12.5 Tonelli–Shanks 算法 . . . 183
4.13 类欧几里得算法 . . . 183

5 集合论 群论 **190**
5.1 集合论定理 . . . 190
5.1.1 模意义统计方案 . . . 191
5.2 拉格朗日定理 . . . 191
5.3 置换群 . . . 191
5.3.1 Burnside 引理 . . . 191
5.3.2 Pólya 定理 . . . 192
5.3.3 常见题型 . . . 192
5.3.4 完全图染色问题 . . . 197
5.3.5 置换的群性质 . . . 199

6 组合数学 **200**
6.1 盒子放球问题总结 . . . 200
6.2 Catalan 数 . . . 201
6.2.1 性质 . . . 201
6.2.2 常见应用 . . . 202
6.2.3 扩展 . . . 203
6.3 Stirling 数 . . . 206
6.3.1 第一类 Stirling 数 . . . 206
6.3.2 第二类 Stirling 数 . . . 207
6.4 贝尔数 . . . 207
6.4.1 因子分解应用 . . . 208
6.4.2 贝尔数计算 . . . 208
6.5 Lucas/ExLucas . . . 210
6.5.1 Lucas 定理 . . . 210
6.5.2 ExLucas . . . 211
6.6 康托展开 . . . 212
6.7 其它公式与定理 . . . 212
6.7.1 常用公式 . . . 212
6.7.2 Raney 引理 . . . 213

7 多项式 **214**
7.1 快速傅里叶变换 FFT . . . 215
7.1.1 FFT 原理 . . . 215
7.1.2 迭代 FFT 实现 . . . 217
7.1.3 实序列 DFT . . . 219
7.1.4 MTT 之拆系数 FFT . . . 220
7.2 快速数论变换 NTT . . . 220
7.2.1 NTT 原理 . . . 220
7.2.2 NTT 实现 . . . 220
7.2.3 NTT 常见模数 . . . 220
7.2.4 MTT 之三模数 NTT . . . 221
7.3 快速沃尔什变换 FWT . . . 221
7.3.1 FWT 原理 . . . 221
7.3.2 nand,nor,nxor . . . 222
7.3.3 FWT 实现 . . . 222
7.4 快速莫比乌斯变换/反演 . . . 224
7.4.1 子集卷积 . . . 226
7.5 多项式高级算法 . . . 226
7.5.1 牛顿迭代法 . . . 226

7.5.2 多项式开方 . . . 227
7.5.3 多项式求逆 . . . 227
7.5.4 多项式取模 . . . 227
7.5.5 多项式求导与积分 . . . 228
7.5.6 多项式 ln . . . 228
7.5.7 多项式 exp . . . 228
7.5.8 多项式快速幂 . . . 228
7.5.9 多项式三角函数 . . . 228
7.5.10 进制转换 . . . 235
7.5.11 多项式多点求值 . . . 235
7.5.12 多项式多点插值 . . . 240
7.5.13 组合数取模 . . . 241
7.5.14 CDQ 分治 FFT . . . 241
7.5.15 多项式乘积(普通分治 FFT) . . . 241
7.5.16 二元多项式卷积 . . . 242
7.5.17 循环卷积 . . . 242
7.5.18 CZT . . . 245
7.6 多项式算法封装 . . . 246
7.7 计算形式幂级数的牛顿迭代法的常数优化 . . . 248
7.7.1 卷积 . . . 248
7.7.2 求逆 . . . 248
7.7.3 平方根 . . . 249
7.7.4 指数 . . . 249
7.7.5 优化方法总结与实现细节 . . . 251
7.7.6 实验结果 . . . 251
7.8 本章注记 . . . 252

8 线性代数 253
8.1 高斯消元 . . . 253
8.1.1 变换为上三角矩阵 . . . 253
8.1.2 求解线性方程组 . . . 254
8.1.3 求解逆矩阵 . . . 254
8.2 LUP 分解 . . . 257
8.2.1 基本原理 . . . 257
8.2.2 LUP 分解 . . . 257
8.2.3 正向/反向替换 . . . 259
8.2.4 LUP 分解求逆矩阵 . . . 259
8.3 行列式 . . . 262
8.3.1 定义与性质 . . . 262
8.3.2 求行列式 . . . 263
8.4 线性基 . . . 263
8.4.1 插入 . . . 263
8.4.2 合并 . . . 263
8.4.3 存在性查询 . . . 263
8.4.4 最大值 . . . 264
8.4.5 最小非 0 值 . . . 264
8.4.6 第 k 小值 . . . 264
8.5 最小二乘逼近 . . . 265
8.6 常系数齐次线性递推 . . . 266

9 数学杂项 271
9.1 泰勒级数展开 . . . 272
9.1.1 形式 . . . 272
9.1.2 常见泰勒级数展开 . . . 272
9.2 Simpson 积分 . . . 273
9.2.1 形式与推导 . . . 273
9.2.2 自适应 Simpson . . . 273
9.3 概率与期望 . . . 274
9.3.1 全概率公式 . . . 274
9.3.2 贝叶斯定理 . . . 274
9.3.3 期望 . . . 274
9.3.4 伯努利试验 . . . 275
9.4 生成函数 . . . 276
9.4.1 普通型生成函数 . . . 276
9.4.2 指数型生成函数 . . . 276
9.4.3 自然数幂求和 . . . 276
9.4.4 生成函数处理背包问题 . . . 277
9.4.5 递推式与生成函数之间的转换 . . . 278
9.4.6 快速求指定数集的幂和 . . . 278
9.4.7 概率生成函数 . . . 278
9.5 拉格朗日插值 . . . 279
9.5.1 原理 . . . 279
9.5.2 插值多项式计算 . . . 279
9.5.3 多点插值 . . . 280
9.5.4 缺点 . . . 280
9.5.5 求解自然数幂和 . . . 280
9.6 反演 . . . 281
9.6.1 反演定义 . . . 281
9.6.2 二项式反演 . . . 281
9.6.3 斯特林反演 . . . 282
9.6.4 子集反演 . . . 283
9.6.5 多重子集反演 . . . 283
9.6.6 最值反演(minmax 容斥) . . . 283
9.6.7 单位根反演 . . . 285
9.6.8 拉格朗日反演 . . . 285
9.7 常见数学公式与应用 . . . 286
9.7.1 常见公式 . . . 286
9.7.2 调和级数应用 . . . 286
9.8 线性时间复杂度整点插值 . . . 287
9.9 Berlekamp–Massey 算法 . . . 288
9.10 生成图计数 . . . 290
9.10.1 基本要点 . . . 291
9.10.2 实例 . . . 291
9.11 特征方程 . . . 292

10 动态规划 **294**
10.1 背包优化 . . . 295
10.1.1 bitset 优化 . . . 295
10.1.2 完全背包优化 . . . 295
10.1.3 多重背包优化 . . . 295
10.2 数位动规 . . . 297
10.3 基于连通性状态压缩的动态规划/轮廓线 DP/插头 DP . . . 297
10.3.1 简单回路问题 . . . 297
10.3.2 简单路径问题 . . . 305
10.3.3 最大化回路/路径/连通块点权和 . . . 305
10.3.4 广义括号表示法 . . . 306
10.3.5 棋盘染色问题 . . . 306
10.3.6 局部连通性加速 DP . . . 306
10.4 单调队列优化 . . . 306
10.5 斜率优化 . . . 307
10.5.1 推导 . . . 307
10.5.2 应用 . . . 307
10.5.3 树上斜率优化 . . . 308
10.5.4 CDQ 分治维护凸包 . . . 309
10.6 四边形不等式优化 . . . 309
10.7 矩阵快速幂优化 . . . 310
10.7.1 常规矩阵快速幂 . . . 310
10.7.2 矩阵链乘秩分解 . . . 313
10.7.3 dp 步伐不一致时的解决方案 . . . 316
10.7.4 矩阵对角化加速快速幂 . . . 316
10.7.5 固定转移矩阵多询问 DP . . . 317
10.8 动态 dp . . . 317
10.8.1 常规树上 DDP 方法 . . . 317
10.8.2 全局平衡二叉树 . . . 322
10.8.3 子树 DP 值查询 . . . 326
10.8.4 固定思路 . . . 327
10.9 决策单调性 DP . . . 327
10.9.1 双指针优化 . . . 327
10.9.2 决策二分栈 DP . . . 327
10.9.3 分治 DP . . . 329
10.9.4 时间复杂度陷阱 . . . 334
10.9.5 决策单调性快速判断 . . . 334
10.10WQS 二分凸优化 . . . 334
10.11特殊 DP . . . 335
10.11.1 LCIS . . . 335
10.11.2 Dilworth 定理 . . . 336
10.11.3 杨氏图表(排序矩阵) . . . 336
10.11.4 GarsiaWachs 算法 . . . 336
10.11.5 双单调性优化 . . . 337
10.11.6 折半状压 DP . . . 337
10.12杂记 . . . 338

11 树 **339**
11.1 最小公共祖先 . . . 339
11.1.1 倍增法 . . . 339
11.1.2 树链剖分 . . . 340
11.1.3 欧拉序 +ST 表 . . . 340
11.1.4 Tarjan . . . 342
11.2 链剖分 . . . 343
11.2.1 轻重链剖分 . . . 343
11.2.2 长链剖分 . . . 345
11.3 树的直径 . . . 348
11.3.1 定义与计算 . . . 348
11.3.2 性质 . . . 348
11.3.3 多子树直径查询 . . . 348
11.4 Dsu On Tree . . . 349
11.5 Purfer Sequence . . . 350
11.5.1 构造 Purfer Sequence . . . 350
11.5.2 恢复原树 . . . 350
11.5.3 计数应用 . . . 350
11.6 虚树 . . . 350
11.7 点分治 . . . 352
11.7.1 静态点分治 . . . 352
11.7.2 动态点分治 . . . 366

12 最短路相关问题 **377**
12.1 单/多源最短路 . . . 377
12.1.1 SPFA . . . 377
12.1.2 算法优化 . . . 378
12.1.3 Dijkstra . . . 378
12.1.4 01 最短路 . . . 378
12.1.5 Johnson 算法 . . . 381
12.2 差分约束系统 . . . 382
12.2.1 与最短/长路的关系 . . . 382
12.2.2 判断是否有可行解 . . . 382
12.2.3 求解 . . . 385
12.3 k 短路 . . . 385
12.3.1 A* 算法 . . . 385
12.3.2 可持久化左偏树法 . . . 385

13 生成树 **391**
13.1 最小生成树 . . . 391
13.1.1 Kruskal 与 LCT . . . 391
13.1.2 动态 MST . . . 392
13.1.3 Prim 算法 . . . 396
13.1.4 次小生成树 . . . 397
13.1.5 生成树性质 . . . 397
13.1.6 Boruvka 算法 . . . 399
13.1.7 单点度限制最小生成树 . . . 402
13.2 Kruskal 重构树 . . . 402
13.2.1 构造 . . . 402
13.2.2 应用 . . . 402

13.3 曼哈顿距离 MST . . . 403
13.3.1 分析 . . . 403
13.3.2 实现 . . . 403
13.4 Matrix-Tree 定理 . . . 405
13.4.1 基本定义 . . . 405
13.4.2 扩展 . . . 405
13.5 斯坦纳树 . . . 407

14 图论 **409**
14.1 割点与桥 . . . 410
14.1.1 割点 . . . 410
14.1.2 桥 . . . 412
14.2 强连通分量 . . . 412
14.2.1 定义 . . . 412
14.2.2 Tarjan 算法 . . . 413
14.3 双连通分量 . . . 414
14.3.1 点双连通分量 . . . 414
14.3.2 边双连通分量 . . . 414
14.4 2-SAT . . . 415
14.4.1 问题描述 . . . 415
14.4.2 可行性判定 . . . 415
14.4.3 构造方案 . . . 415
14.4.4 前后缀优化建图 . . . 416
14.5 仙人掌与圆方树 . . . 416
14.5.1 仙人掌 . . . 416
14.5.2 圆方树 . . . 418
14.5.3 广义圆方树 . . . 420
14.6 图上路径 . . . 421
14.6.1 欧拉路径与欧拉回路 . . . 421
14.6.2 哈密尔顿回路 . . . 424
14.7 弦图 . . . 424
14.7.1 相关概念 . . . 424
14.7.2 弦图判定 . . . 425
14.7.3 弦图的极大团 . . . 428
14.7.4 弦图的点染色 . . . 428
14.7.5 弦图的最大独立集与最小团覆盖 . . . 430
14.7.6 区间图 . . . 430
14.8 带花树 . . . 430
14.8.1 无向图最大匹配 . . . 430
14.8.2 Micali-Vazirani Algorithm . . . 434
14.8.3 近线性复杂度的随机匹配算法 . . . 434
14.8.4 无向图最大权匹配 . . . 436
14.9 支配树 . . . 436
14.9.1 定义 . . . 436
14.9.2 DAG 的支配树 . . . 436
14.9.3 一般图的支配树 . . . 436
14.10杂讲 . . . 436
14.10.1 竞赛图 . . . 436
14.10.2 最小平均值环 . . . 437
14.10.3 平面图性质 . . . 437

14.10.4 拓扑排序判环 437
14.10.5 Lindström–Gessel–Viennot Lemma 438
14.10.6 三元环计数 439

15 字符串 **440**
15.1 Hash 441
15.1.1 BKDRHash 441
15.1.2 混合 Hash 441
15.1.3 子串 Hash 441
15.2 Trie 字典树 442
15.3 AC 自动机 442
15.3.1 构造 442
15.3.2 查询 443
15.3.3 fail 树 444
15.4 Huffman 编码 444
15.4.1 常规 Huffman 444
15.4.2 k 叉 Huffman 444
15.4.3 效率优化 445
15.4.4 限长 Huffman 445
15.4.5 编码字符代价不等的 Huffman 447
15.5 KMP 算法 448
15.6 Manacher 算法 448
15.7 回文自动机 450
15.7.1 构造 450
15.7.2 应用 451
15.8 后缀树 452
15.8.1 构造 452
15.8.2 广义后缀树 452
15.8.3 应用 452
15.9 后缀数组 453
15.9.1 倍增构造 453
15.9.2 最长公共前缀 454
15.9.3 应用 455
15.10后缀仙人掌 457
15.10.1 概述 457
15.10.2 构造 457
15.10.3 应用 457
15.11后缀自动机 458
15.11.1 描述 458
15.11.2 构造 458
15.11.3 Parent 树的应用 460
15.11.4 线性构造后缀数组 464
15.11.5 SAM with LCT 466
15.11.6 广义 SAM 466
15.11.7 序列自动机 467
15.12算术表达式解析 468
15.13Z Algorithm 470
15.13.1 求解 470
15.13.2 应用 471
15.14卷积法解决字符串匹配问题 471

15.14.1 回文子序列 . . . 471
15.14.2 带通配符匹配 . . . 475
15.14.3 大字符集处理 . . . 475
15.14.4 广义模式匹配 . . . 477
15.15 最小表示法 . . . 477

16 计算几何 **479**
16.1 基础设施 . . . 480
16.1.1 点, 向量, 直线, 半平面的表示 . . . 480
16.1.2 点乘与叉乘 . . . 480
16.1.3 点到直线的距离 . . . 481
16.1.4 直线、线段的交点 . . . 481
16.1.5 判定点是否在多边形内 . . . 481
16.1.6 向量的旋转 . . . 482
16.1.7 坐标系的切换 . . . 482
16.1.8 点、向量、法向量的坐标变换 . . . 482
16.1.9 反射与折射 . . . 483
16.1.10 pick 定理 . . . 484
16.1.11 切比雪夫距离 . . . 484
16.1.12 精度处理 . . . 484
16.2 凸包 . . . 485
16.2.1 极角序凸包 . . . 485
16.2.2 水平序凸包 . . . 485
16.2.3 在线凸包 . . . 486
16.2.4 凸包矢量和(闵可夫斯基和) . . . 486
16.2.5 凸包合并 . . . 486
16.2.6 稀疏包分布 . . . 486
16.2.7 二维最小乘积生成树 . . . 487
16.2.8 三维凸包 . . . 487
16.2.9 快速凸包 . . . 490
16.3 圆 . . . 495
16.3.1 圆的并 . . . 495
16.3.2 圆的交 . . . 502
16.3.3 最小圆覆盖 . . . 502
16.3.4 圆的反演 . . . 505
16.4 半平面交 . . . 505
16.4.1 基本算法 . . . 505
16.4.2 线性判空集 . . . 508
16.5 旋转卡壳 . . . 508
16.6 平面最近点对 . . . 509

17 最优化问题 **512**
17.1 最优化方法 . . . 512
17.1.1 牛顿迭代法 . . . 512
17.1.2 爬山法 . . . 513
17.1.3 模拟退火 . . . 513
17.1.4 遗传算法 . . . 514
17.1.5 A* . . . 514
17.1.6 梯度下降 . . . 514
17.1.7 解集存储 . . . 515

17.1.8 拉格朗日乘子法 . . . 515
17.2 01 分数规划 . . . 517
17.2.1 二分答案法 . . . 517
17.2.2 Dinkelbach 法 . . . 518
17.3 线性规划 . . . 518
17.3.1 定义与规范描述形式 . . . 518
17.3.2 单纯形算法 . . . 519
17.3.3 对偶线性规划 . . . 525
17.3.4 全幺模矩阵 . . . 526
17.4 随机化算法 . . . 526
17.4.1 Monte Carlo 算法 . . . 526
17.4.2 Las Vegas 算法 . . . 527

18 理论 **528**
18.1 时间复杂度分析 . . . 528
18.1.1 主定理 . . . 528
18.1.2 Akra-Bazzi 法 . . . 529
18.2 数值编码 . . . 529
18.2.1 整数编码 . . . 529
18.2.2 浮点数编码 . . . 530
18.3 常见排序算法 . . . 530
18.3.1 比较排序算法 . . . 530
18.3.2 非比较排序算法 . . . 530
18.4 NP 完全性 . . . 530
18.4.1 定义 . . . 530
18.4.2 常见 NPC 问题 . . . 531

19 杂项 **532**
19.1 思路与技巧 . . . 533
19.1.1 二分/三分 . . . 533
19.1.2 补集转化 . . . 533
19.1.3 莫队 . . . 533
19.1.4 分块 . . . 540
19.1.5 MITM . . . 542
19.1.6 倍增 . . . 543
19.1.7 随机化 . . . 543
19.1.8 按位拆分 . . . 543
19.1.9 扫描线 . . . 543
19.1.10 差分 . . . 543
19.1.11 双指针法 . . . 543
19.1.12 优先队列维护长序列 . . . 544
19.1.13 集合选数最值问题 . . . 544
19.1.14 二进制分组 . . . 546
19.1.15 整体二分 . . . 547
19.1.16 cdq 分治 . . . 547
19.1.17 Kernelization . . . 548
19.1.18 启发式合并 . . . 548
19.1.19 启发式分治 . . . 548
19.1.20 分段打表 . . . 548
19.1.21 注意事项/常见转化/思想 . . . 548

19.1.22 比赛注意事项 . . . 552
19.1.23 本节注记 . . . 554
19.2 卡常 . . . 554
19.2.1 取模 . . . 554
19.2.2 矩阵乘法 . . . 554
19.2.3 基于硬件的优化 . . . 557
19.2.4 位运算 . . . 558
19.2.5 搜索优化 . . . 560
19.2.6 数组清零 . . . 560
19.2.7 读入优化 . . . 560
19.2.8 快速乘法取模 . . . 563

20 总结 **564**
20.1 基于输入 . . . 564
20.1.1 单变量 . . . 564
20.1.2 元素序列 . . . 564
20.1.3 时间/区间操作序列 . . . 565
20.1.4 树 . . . 565
20.1.5 图 . . . 565
20.1.6 字符串 . . . 565
20.2 基于关键字/特征 . . . 566
20.3 时间轴有关问题 . . . 566
20.3.1 静态问题 . . . 566
20.3.2 动态问题 . . . 567
20.3.3 时间分治技巧 . . . 567
20.4 树上连通块问题 . . . 567
20.4.1 无色连通块 . . . 568
20.4.2 黑白连通块 . . . 568

A 资料推荐 **569**
A.1 个人博客 . . . 569
A.2 好用的网站/工具 . . . 569
A.3 优秀资料 . . . 570
A.4 推荐书籍 . . . 570
A.5 其它 . . . 571

B Index **572**

C Bibliography **576**

后记 **577**
C.1 初稿 . . . 577
C.2 二轮复习 . . . 578
C.3 三轮复习 . . . 580
C.4 FCS NOI2019 随笔 . . . 581
C.4.1 day0 . . . 581
C.4.2 day1 . . . 582
C.4.3 day1 下午 . . . 582
C.4.4 day2 . . . 583
C.5 End . . . 583

Chapter 1

博弈论

1.1 SG 函数与 SG 定理 1
1.1.1 适用范围 1
1.1.2 SG 函数 2
1.1.3 SG 定理 3
1.2 Nim 系列游戏 5
1.2.1 Nim 游戏 5
1.2.2 Bash 游戏 5
1.2.3 NimK 游戏 5
1.2.4 Anti Nim 6
1.2.5 阶梯博弈 6
1.2.6 威佐夫博弈 7
1.2.7 斐波那契博弈 8
1.3 Alpha-Beta 剪枝 9
1.3.1 Min-Max 搜索 9
1.3.2 Alpha-Beta 剪枝 11
1.4 本章注记 11

1.1 SG 函数与 SG 定理

1.1.1 适用范围

一切 Impartial Combinatorial Games 都等价于 Nim 游戏, 可以使用 SG 函数解决。该类游戏拥有如下特征:[1]

- 两个玩家轮流操作
- 当有一名玩家无法操作时, 游戏结束
- 游戏会在有限次操作后结束(状态转移图是一个 DAG)
- 游戏对双方是公平的, 所有操作必须能够由双方完成(即当双方都采取最优策略时, 游戏的胜负只取决于先后手)
- 双方在开局前已知道关于游戏的全部信息, 并在游戏时采用最优策略。

[1]参见 Impartial game - Wikipedia https://en.wikipedia.org/wiki/Impartial_game

1.1.2 SG 函数

接下来给出必胜点和必败点的定义(前提是双方均采用最优策略):

- 必胜点(N-Position):处于该状态的玩家必胜
- 必败点(P-Position):处于该状态的玩家必败

必胜点与必败点有如下性质:

- **性质 1.1** 终结点为必败点
- **性质 1.2** 必败点的下一状态必然为必胜点(某玩家必败,等价于无论他如何操作都使另一玩家必胜)
- **性质 1.3** 从必胜点出发至少有一种方式进入必败点(该玩家的最优策略就是使状态转移到必败点)

要判断哪个玩家必胜,一般使用 SG 函数和 SG 定理计算出先手所在状态(即初始状态)是必胜点还是必败点。

SG 函数的定义如下:$SG(x) = mex(S(x))$

其中 $S(x)$ 是状态 x 的后继状态的 SG 函数值的集合,$mex(S)$ 是没有出现在集合 S 中的最小非负整数。

以 Nim 游戏为例,可根据函数定义计算 SG 值:

NimSG

```
#include <cstdio>
#include <cstring>
const int size = 25;
int SG[size];
int getSG(int x) {
    if(SG[x] == -1) {
        bool flag[size] = {};
        for(int i = 0; i < x; ++i)
            flag[getSG(i)] = true;
        for(int i = 0; i <= x; ++i)
            if(!flag[i]) {
                SG[x] = i;
                break;
            }
    }
    return SG[x];
}
int main() {
    memset(SG, -1, sizeof(SG));
    for(int i = 0; i <= 20; ++i)
        printf("SG(%d)=%d\n", i, getSG(i));
    return 0;
}
```

1.1.3 SG 定理

定理 1.4 (Sprague-Grundy Theorem A) 当某玩家无法操作时，认为该玩家失败。若 $SG(x)=0$，则该状态为必败态，否则为必胜态。

归纳证明：假设该定理对状态 x 后继的状态成立，则

- 若 $SG(x)>0$，则说明存在一个后继状态 y，使得 $SG(y)=0$，因为 y 为必败态，所以 x 为必胜态。
- 若 $SG(x)=0$，则说明对 x 的任意后继状态 y，都有 $SG(y)>0$，因为 y 为必胜态，所以 x 为必败态。

定理 1.5 (Sprague-Grundy Theorem B) 游戏的 SG 函数值等于各子游戏函数值的 Nim 和（即 xor 和）。

设 S_X 为 X 后继状态的集合，$b=SG(X_1)\oplus\cdots\oplus SG(X_N)$。
该定理可分为两个引理证明：

1. **引理 1.6** $\forall_{a\in N,a<b},\exists_{X'\in S_X},SG(X')=a$
   归纳证明：首先假设该引理对子游戏成立。
   设 $d=b\oplus a$，d 的最高位为 k，则存在 $SG(X_i)$ 的第 k 位为 1（d 的那一位由奇数个 $SG(X_i)$ 贡献）。
   所以 $SG(X_i)\oplus d<SG(X_i)$，由假设得 $\exists_{X_i'\in S_{X_i}},SG(X_i')=SG(X_i)\oplus d$。
   结合 $a=b\oplus d$ 得 $a=SG(X_1)\oplus\cdots\oplus SG(X_i')\oplus\cdots\oplus SG(X_N)$
   又因为 $\{X_1,\cdots,X_i',\cdots,X_N\}\in S_X$，所以引理 1.6得证。
2. **引理 1.7** $\forall_{X'\in S_X},SG(X')\neq b$
   反证法：假设 SG 定理对子状态成立（这里貌似不严谨），
   且 $\exists_{X'\in S_X},SG(X')=b$。
   那么就有 $SG(X_i')=SG(X_i)$，由 mex 函数的定义可得两式矛盾，引理 1.7得证。

以上内容参考了 Angel_Kitty[2]与 PhilipsWeng[3]的博客。

1.1.3.1 例题

Luogu P2575 高手过招[4]
每行棋子可视为一个子游戏，状压后使用 DFS 计算 SG 函数值，然后利用定理 1.5计算整个游戏的 SG 函数值。

Luogu P2575

```
0 #include <cstdio>
  #include <cstring>
  int read(){
      int res=0,c;
      do c=getchar();
```

[2]SG 函数和 SG 定理 [详解] Angel_Kitty https://www.cnblogs.com/ECJTUACM-873284962/p/6921829.html
[3]SG 定理 https://blog.csdn.net/PhilipsWeng/article/details/48395375
[4]https://www.luogu.org/problemnew/show/P2575

```
    while(c<'0'||c>'9');
    while('0'<=c&&c<='9'){
        res=res*10+c-'0';
        c=getchar();
    }
    return res;
}
int sg[1<<20];
int calcSg(int st){
    if(sg[st]==-1){
        bool flag[20]={};
        for(int i=0;i<20;++i){
            int cp=1<<i;
            if(st&cp){
                int clear=~((cp<<1)-1);
                int mask=(~(st&clear))&clear;
                int pos=mask&-mask;
                if(pos<(1<<20)){
                    int dst=st^cp^pos;
                    flag[calcSg(dst)]=true;
                }
            }
        }
        for(int i=0;i<20;++i)
            if(!flag[i]){
                sg[st]=i;
                break;
            }
        }
    return sg[st];
}
bool foo(){
    int n=read(),res=0;
    while(n--){
        int m=read();
        int st=0;
        while(m--)st|=1<<(read()-1);
        res^=calcSg(st);
    }
    return res;
}
int main(){
    memset(sg,-1,sizeof(sg));
    int t=read();
    while(t--)puts(foo()?"YES":"NO");
    return 0;
}
```

1.2 Nim 系列游戏

1.2.1 Nim 游戏

普通 Nim 游戏的定义:有两个玩家轮流从许多堆中移除对象。在每个回合中, 玩家选择一个非空的堆, 可以移除任意数量的对象, 但要移除至少一个对象。无法操作的玩家为败者。

此类游戏可看做是 Bash 游戏的特殊化。

定理 1.8 $SG_{Nim}(x) = x$

证明略。

1.2.2 Bash 游戏

Bash 游戏与普通 Nim 游戏的区别是增加了每次最多移除 k 个对象的限制。

定理 1.9 $SG_{Bash}(x) = x \bmod (k+1)$

证明略。

1.2.3 NimK 游戏

NimK 游戏与普通 Nim 游戏的区别是每次可以从不超过 k 个堆中移除任意数目对象。

定理 1.10 此状态为必败态当且仅当将每堆对象的数目拆位, 每位上 *1* 的个数 $S[i] \bmod (k+1)$ 均为 *0*。

记忆:普通 Nim 游戏可理解为 mod 2 的情况。
算法正确性证明:
当前玩家可以通过如下步骤赢得游戏:
策略是不断地转移到每位上 1 的个数 $\bmod(k+1)$ 均为 0 的状态。
设 $D0[i]$ 与 $D1[i]$ 为已标记堆中第 i 位为 0 和 1 的个数。

1. 选取 $S[i]$ 非 0 最高位 W;
2. 找到一个第 W 位为 1, 且未标记的堆, 将该堆标记, 把它的第 W 位改为 0, 并更新 $D0[1 \sim W-1], D1[1 \sim W-1]$;
3. 如果 $S[i]$ 非 0, 且 $S[i] + D0[i] > k || S[i] - D1[i] < 1$, 则可以通过修改已标记的堆将 $S[i]$ 变为 0。
4. 如果 $S[i]$ 均为 0, 则结束循环, 否则重复步骤 1。

模 $k+1$ 保证了修改的堆数不超过 k, 因为如果已标记了 k 个堆, $S[i] + D0[i] > k || S[i] - D1[i] < 1$ 必定成立, 无需再标记新堆。

所以当每位均为 0 时, 当前玩家要么已经无法操作, 要么必须转移至必胜态(对方可根据上述方法转移回必败态), 因此该状态为必败态。

定理 1.10得证。

1.2.4 Anti Nim

不能操作的玩家胜利。

定理 1.11 先手必胜当且仅当满足以下条件之一:

1. $SG(x) = 0$ 且所有堆的对象数都为 *1*
2. $SG(x) \neq 0$ 且至少有一堆对象数大于 *1*

证明:定义对象数为 1 的叫 A 堆, 大于 1 的叫 B 堆。

1. 若所有堆均为 A 堆, 则奇数堆先手必败, 反之必胜。
2. 若 B 堆数等于 1, 显然 $SG(x) \neq 0$, 则可根据堆的总数确定取完该堆还是剩 1 个, 使下一状态为情况 1 的必败态, 所以先手必胜。
3. 若 B 堆数大于 1, 则
   (a) 若 $SG(x) = 0$, 则必须留下超过 2 个 B 堆并使 $SG(x') \neq 0$, 否则会使对方进入情况 2 的必胜态。
   (b) 若 $SG(x) \neq 0$, 则根据 Nim 游戏的理论(必胜态 $\rightarrow$ 必败态), 存在一种方法转移至情况 3 的子情况 (a)。

   若玩家处于情况 3 的子情况 (b) 中, 则可以在有限次回合内使对方无法转移至子情况 (b), 因此该状态为必胜态。

定理 1.11得证。

1.2.5 阶梯博弈

阶梯博弈的定义:有多个阶梯, 从左到右编号为 $1 \sim n$, 1 号阶梯的左边为地面, 阶梯上有若干石子, 玩家每次可以将某阶梯的石子移动至其左边的下一级阶梯, 当某玩家无法移动石子时(即所有石子都在地面), 该玩家失败。

定理 1.12 阶梯博弈问题等价于奇数号阶梯的 *Nim* 博弈。

证明:假设己方是先手:

1. 当对方移动奇数号阶梯的石子到偶数号阶梯时, 我们按照 Nim 游戏的策略从奇数号阶梯移动石子到偶数号阶梯(等价于移除石子)。
2. 当对方移动偶数号阶梯的石子到奇数号阶梯时, 我们将其移动的石子移动到偶数号阶梯, 抵消对方的操作。

可以将移动到偶数号阶梯看做被移除(最终移动到地面)。使用上述策略可以使状态与偶数号阶梯的石子数无关, 定理 1.12得证。

1.2.5.1 例题

Luogu P3480 [POI2009]KAM-Pebbles[5]
将原条件做差分变换可将此题转换为阶梯博弈模型($A_i \geq A_{i-1} \Leftrightarrow A_i - A_{i-1} \geq 0$)。

Luogu P3480

```
#include <cstdio>
int read(){
    int res=0,c;
    do c=getchar();
    while(c<'0'||c>'9');
    while('0'<=c&&c<='9'){
        res=res*10+c-'0';
        c=getchar();
    }
    return res;
}
bool foo(){
    int n=read();
    int res=0,last=0,mask=n&1;
    for(int i=1;i<=n;++i){
        int val=read();
        if((i&1)==mask)res^=val-last;
        last=val;
    }
    return res;
}
int main(){
    int t=read();
    while(t--)puts(foo()?"TAK":"NIE");
    return 0;
}
```

出题灵感:Anti BashK 游戏

1.2.6 威佐夫博弈

有两堆石子, 玩家可从一个堆中拿出石子或者从两个堆中拿出相同数量的石子, 不能操作者输。

我们可以使用 SG 定理写一个打表程序:

```
#include <cstdio>
#include <cstring>
const int size = 21;
int sg[size][size];
int SG(int a, int b) {
    if(a == 0 && b == 0)
        return 0;
    if(sg[a][b] == -1) {
        bool flag[1000] = {};
```

[5] https://www.luogu.org/problemnew/show/P3480

```
            for(int i = 1; i <= a; ++i)
                flag[SG(a - i, b)] = true;
            for(int i = 1; i <= b; ++i)
                flag[SG(a, b - i)] = true;
            for(int i = 1; i <= a && i <= b; ++i)
                flag[SG(a - i, b - i)] = true;
            for(int i = 0; i < 1000; ++i)
                if(!flag[i]) {
                    sg[a][b] = i;
                    break;
                }
        }
        return sg[a][b];
    }
    int main() {
        memset(sg, -1, sizeof(sg));
        for(int i = 0; i < size; ++i) {
            for(int j = 0; j < size; ++j) {
                printf("%d ", SG(i, j));
            }
            puts("");
        }
        for(int i = 0; i < size; ++i)
            for(int j = i; j < size; ++j)
                if(SG(i, j) == 0)
                    printf("%d %d\n", i, j);
        return 0;
    }
```

打印出 20 以内的必败点:

0 0
1 2
3 5
4 7
6 10
8 13
9 15
11 18
12 20

可以发现每一个必败点的石子数满足较小数是之前未出现过的最小自然数, 较大数与较小数之差递增。进一步研究可以发现两数之差与较小数之比近似等于黄金分割比。

必败点的局面构成的序列为 Beatty 序列, 两个数列可由两个无理数 α, β 生成, 即 $\{\lfloor \alpha n \rfloor\}, \{\lfloor \beta n \rfloor\}$, 其中 $\frac{1}{\alpha} + \frac{1}{\beta} = 1$。Rayleigh Theorem 描述了 Beatty 序列中任意一个正整数仅在某个数列中出现一次。由两个数列的关系可得 $(\alpha + 1)n = \beta n$, 联立解得 $\alpha = \frac{\sqrt{5}+1}{2}$。因此只要一行代码就可以判断必败点: $a == static\_cast < int > ((b - a) * 1.618 \cdots)$。

1.2.7 斐波那契博弈

仅一堆石子, 先手不能把所有石子取完, 至少取 1 颗。每一步玩家可以取的石子数不超过上一次取的石子数的 2 倍, 取出最后一颗石子的玩家胜利。

定理 1.13 当且仅当石子数为斐波那契数时，该局面为必败态。

证明 首先证明当石子数为斐波那契数时，该局面为必败态。

记 f_i 为第 i 个斐波那契数。显然当 $i \leq 2$ 时，该局面为必败态。使用归纳法，假设对于 $i < k$，该结论成立。当 $i = k$ 时，此时将石子看成两部分，即 $f_k = f_{k-1} + f_{k-2}$。由于 $f_{k-1} < 2f_{k-2}$，先手肯定不能取完 f_{k-2} 的部分。由假设可知后手取得了 f_{k-2} 部分的最后一颗石子，考虑后手此时取的石子数。由贪心（后手要刚好取完 f_{k-2} 部分）可知当先手第一次取的石子数 $\geq \frac{f_{k-2}}{3}$ 时，后手取的石子数 $\leq \frac{2f_{k-2}}{3}$，此时后手取完 f_{k-2} 部分后，先手最多取得 $\frac{4f_{k-2}}{3} < f_{k-1}$，即先手仍然不能一次性取完 f_{k-1} 部分，由假设可知先手必败。

然后证明当石子数不为斐波那契数时，该局面为必胜态。

引理 1.14 (Zeckendorf's Theorem) 任意正整数可分解为多个不连续的斐波那契数之和。

每次用尽可能大的斐波那契数将石子拆分为多个部分。先手可将数量最小的部分取完，由于斐波那契数的不连续性，后手不可能取完倒数第二部分的石子。此时后手处于这一部分石子的必败态，即先手将取得这部分石子的最后一颗。对于每一堆都如此操作就能赢得胜利。

以上内容参考了 forezxl[6]，hehedad[7]，我爱 AI_AI 爱我[8] 和 dgq8211[9]的博客，以及百度百科(终于发现比维基更全的词条了)[10]。

1.3 Alpha-Beta 剪枝

1.3.1 Min-Max 搜索

核心在于搜索出对双方来说最好的操作。在实践中可以将对手的最优估价函数取反，这样可以避免写区分双方的复杂代码。

伪代码如下：

Min-Max Search

```
0    int DFS(State s) {
         if(s.end())return s.eval();
         else {
             int res=-inf;
             for(auto nxt:s)
                 res = std::max(res,-DFS(nxt));
             return res;
         }
     }
```

[6]anti-Nim 游戏(反 Nim 游戏)简介 https://blog.csdn.net/a1799342217/article/details/78274410

[7]关于 nimk 类型博弈的详细理解与解释 https://blog.csdn.net/chenshibo17/article/details/79783523

[8]阶梯博弈算法详解 https://blog.csdn.net/qq_30241305/article/details/51956518

[9]斐波那契博弈（Fibonacci Nim）- nyist_xiaod https://blog.csdn.net/dgq8211/article/details/7602807

[10]威佐夫博弈 _ 百度百科 https://baike.baidu.com/item/

1.3.1.1 例题

Luogu P4363 [九省联考 2018] 一双木棋 chess [11]
状压 Min-Max 搜索, 需要记忆化。

Luogu P4363

```cpp
#include <cstdio>
#include <map>
const int size = 11, inf = 1 << 30;
int n, m, cur[size], A[2][size][size];
long long end = 0;
typedef std::map<long long, int> Map;
typedef Map::iterator IterT;
Map cache;
#define asInt64(x) static_cast<long long>(x)
int dfs(long long s, bool now) {
    if(s == end)
        return 0;
    {
        IterT it = cache.find(s);
        if(it != cache.end())
            return it->second;
    }
    int res = -inf;
    for(int i = 1; i <= n; ++i)
        if(cur[i - 1] > cur[i]) {
            ++cur[i];
            long long dst =
                s ^ (asInt64((cur[i] - 1) ^ cur[i])
                     << (4 * i));
            res = std::max(res, A[now][i][cur[i]] -
                                dfs(dst, !now));
            --cur[i];
        }
    return cache[s] = res;
}
int main() {
    scanf("%d%d", &n, &m);
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j)
            scanf("%d", &A[0][i][j]);
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j)
            scanf("%d", &A[1][i][j]);
    for(int i = 1; i <= n; ++i)
        end |= asInt64(m) << (4 * i);
    cur[0] = m;
    for(int i = 1; i <= n; ++i)
        cur[i] = 0;
```

[11] https://www.luogu.org/problemnew/show/P4363

```
    printf("%d\n", dfs(0, false));
    return 0;
}
```

1.3.2 Alpha-Beta 剪枝

Min-Max 搜索运行时需要检查整棵博弈树, 而 Alpha-Beta 能够抛弃掉一些不可能为解的子树, 提高搜索速度。核心思想是当你可以明确这个状态不可能被达到时, 就不需要再搜下去了。

设 α 为当前搜索到的下界, β 为上界。

伪代码如下:

Alpha-Beta 剪枝

```
0   int DFS(State s,int alpha,int beta) {
        if(s.end())return s.eval();
        else {
            for(auto nxt:s) {
                int val = -DFS(nxt,-beta,-alpha);// 1
                if(val >= beta)// 2
                    return beta;
                if(val > alpha)
                    alpha = val;
            }
10          return alpha;
        }
    }
    int res = DFS(init,-inf,inf);
```

Alpha-Beta 剪枝与 Min-Max 搜索的区别有二:

1. 对于对手来说, 权值是相反数, 因此 $\alpha' = -\beta, \beta' = -\alpha$。
2. val 会更新 α 的值, 所以 $val >= \beta$ 等价于 $\alpha > \beta$, 已没有继续搜索的必要。对手要么只能接受 $-\beta$, 要么可以避开这条路径(因为 β 值是由上一层节点递归计算同级节点得到的结果, 而走哪条路径的决定权在上级, 即对手)。

Alpha-Beta 的效率严重依赖于搜索顺序:

- 在最坏情况下(无法剪枝), 该算法与 Min-Max 搜索相同, 效率极低。
- 在最优情况下(每次都从最优着法开始搜索), 每层的搜索分支达到 $\sqrt{n}$, 因此搜索深度可以达到原来的两倍。

分支规模大小待证明。

以上内容参考了象棋百科全书[12]

1.4 本章注记

更多博弈论模型参见 博弈游戏的各种经典模型（备忘） - Randolph87 - 博客园 http://www.cnblogs.com/Randolph87/p/5804798.html, 待补充。

[12]计算机博弈 - 象棋百科全书 http://www.xqbase.com/computer.htm

Chapter 2

网络流

2.1 二分图 . . . 13
2.1.1 二分图判定 . . . 13
2.1.2 二分图最大匹配 . . . 14
2.1.3 二分图最大权匹配 Kuhn-Munkras Algorithm . . . 16
2.1.4 二分图常见模型 . . . 17
2.1.5 Hall 定理 . . . 18
2.2 最大流 . . . 20
2.2.1 Dinic 算法 . . . 20
2.2.2 ISAP 算法 . . . 29
2.2.3 HLPP 算法 . . . 31
2.2.4 最大流与最小割 . . . 38
2.2.5 无向图最小割 . . . 38
2.2.6 最小割性质 . . . 42
2.3 费用流 . . . 42
2.3.1 使用 Dijkstra 实现费用流 . . . 42
2.3.2 多路增广费用流 . . . 43
2.3.3 消圈定理 . . . 45
2.3.4 zkw 费用流 . . . 45
2.4 带上下界网络流 . . . 47
2.4.1 无源汇有上下界可行流 . . . 47
2.4.2 有上下界带源汇点可行流 . . . 47
2.4.3 有上下界带一组无限流量源汇点可行流 . . . 48
2.5 常见网络流/最小割模型 . . . 48
2.5.1 平面图转对偶图 . . . 48
2.5.2 最大权闭合子图 . . . 50
2.5.3 区间 k 覆盖问题 . . . 53
2.5.4 集合划分问题 . . . 53
2.6 最小割树 . . . 54
2.6.1 Gusfield 算法构造 . . . 54
2.6.2 询问 . . . 54
2.7 技巧总结 . . . 58
2.7.1 最大流 . . . 58

2.7.2 最小割 58
2.7.3 费用流 58

2.1 二分图

2.1.1 二分图判定

性质 2.1 二分图中不存在奇环。

如果存在奇环, 则必有一条边的端点属于同一集合。所以可以使用 DFS 染色来判定二分图, 遇到矛盾则退出。

BGJudge.cpp

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 105;
struct Edge {
    int to, nxt;
} E[size * size];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
bool vis[size], col[size];
bool DFS(int u, bool c) {
    if(vis[u])
        return col[u] == c;
    vis[u] = true;
    col[u] = c;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(!DFS(v, !c))
            return false;
    }
    return true;
}
int main() {
```

```
    int n = read();
    int m = read();
    while(m--) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    bool flag = true;
    for(int i = 1; i <= n && flag; ++i)
        if(!vis[i])
            flag &= DFS(i, false);
    puts(flag ? "Yes" : "No");
    return 0;
}
```

2.1.2 二分图最大匹配

2.1.2.1 匈牙利算法

匈牙利算法的主要步骤就是遍历左集合的每一个顶点, 使得其尽可能找到一个匹配。要为该顶点找到一个匹配, 首先遍历边, 如果右顶点已经有匹配, 则递归尝试让该匹配点重新找一个匹配, 如果右顶点无匹配或者更换匹配成功, 则这条边是一个匹配。

原则: 有机会上, 没机会创造机会也要上。[1]

感性的算法的正确性证明: 每次递归时匹配数只增不减, 且递归有权修改整个连通块的着色情况。(似乎并没有什么说服力)。

匈牙利算法的时间复杂度为 $O(VE)$, 每次尝试匹配的复杂度为 $O(E)$。

2.1.2.2 Greedy Matching

可以先遍历一次图, 贪心地连边, 以减少尝试拆开匹配边的次数。在图很大的时候有加速效果。

该方法参考了江任捷的演算法筆記[2]。

2.1.2.3 Hopcroft-Karp Algorithm

暂时先坑着~~为什么不写 Dinic 呢~~。

2.1.2.4 例题

Luogu P1129 [ZJOI2007] 矩阵游戏 [3]

首先用二分图最大匹配找到 n 个不同行且不同列的黑格子(置换矩阵 P), 然后就可以操作得到目标矩阵(单位矩阵 I)了。

Luogu P1129

```
#include <cstdio>
#include <cstring>
```

[1] Dark_Scope 趣写算法系列之--匈牙利算法 https://blog.csdn.net/dark_scope/article/details/8880547

[2] 演算法筆記 - Matching http://www.csie.ntnu.edu.tw/~u91029/Matching.html#4

[3] https://www.luogu.org/problemnew/show/P1129

```
const int size=205;
int read(){
    int res=0,c;
    do c=getchar();
    while(c<'0'||c>'9');
    while('0'<=c&&c<='9'){
        res=res*10+c-'0';
        c=getchar();
    }
    return res;
}
struct Edge{
    int to,next;
} E[size*size+10];
int last[size],cnt;
void addEdge(int u,int v){
    ++cnt;
    E[cnt].to=v,E[cnt].next=last[u];
    last[u]=cnt;
}
bool flag[size];
int pair[size];
bool match(int u){
    for(int i=last[u];i;i=E[i].next){
        int v=E[i].to;
        if(!flag[v]){
            flag[v]=true;
            if(!pair[v] || match(pair[v])){
                pair[v]=u;
                return true;
            }
        }
    }
    return false;
}
const char* foo(){
    int n=read();
    cnt=0;
    memset(last,0,sizeof(last));
    for(int i=1;i<=n;++i)
        for(int j=1;j<=n;++j)
            if(read())addEdge(j,i);
    memset(pair,0,sizeof(pair));
    for(int i=1;i<=n;++i){
        memset(flag,0,sizeof(flag));
        if(!match(i))return "No";
    }
    return "Yes";
}
int main(){
```

```
    int t=read();
    for(int i=0;i<t;++i)
        puts(foo());
    return 0;
}
```

2.1.3 二分图最大权匹配 Kuhn-Munkras Algorithm

~~先用费用流做吧~~, ~~暂时先坑着~~。

2.1.3.1 起步

维护每个左/右顶点的权值(称为顶标), 所有节点的顶标和为答案上界。令每个左顶点的顶标为出边边权最大值, 右顶点顶标为 0。

对每个顶点运行匈牙利算法, 若左右顶点顶标之和等于边权, 则考虑连边; 若无法为当前点找到匹配, 则将访问到的左顶点顶标-1, 右顶点顶标 +1, 等价于使答案上界-1(DFS 访问树中的叶子必为左顶点), 重新为该点寻找匹配。把任意二分图当做完全二分图(不存在的边权值为 0), 迭代必定会结束。

这种做法能够保证在找到最大匹配的情况下使权值和最大。

2.1.3.2 优化 1

可以发现在左-1 右 +1 后, 原先边权等于左右顶点顶标之和的边仍然被经过, 一个简单的思路是一次性突破 "瓶颈", 即令下次增广时终点位置处的某条边从不可连边变为可连边, 每次 DFS 增广时维护(顶标和-边权)的最小值 d, 若匹配失败则左 $-d$ 右 $+d$。

这才是复杂度比较靠谱的算法($O(n^3)$)。

2.1.3.3 优化 2

在匹配每个点时, 初始化所有右顶点的松弛函数 $slack$ 为 ∞, 然后 DFS 时 $slack$ 维护(顶标和-边权)的最小值。若匹配失败则令 d 为未访问右顶点的 $slack$ 函数最小值, 左 $-d$ 右 $+d$, 同时未访问节点的 $slack-=d$。

该优化的复杂度不变, 但实测该方法比优化 1 的效率更高(3x)。

2.1.3.4 优化 3

考虑记录其增广时的路径, 然后将递归算法转换为非递归算法。

```
int w[size][size],lh[size],rh[size],pair[size],
    pre[size],slack[size];
bool flag[size];
void aug(int s) {
    reset(flag);
    reset(pre);
    reset(slack,0x3f);
    pair[0]=s;
    int u=0;
    do {
        int v=pair[u],minh=inf,nxt;
        flag[u]=true;
        // 再次DFS后新访问到了点u和它的匹配点
```

```
        // 为点v找新匹配点
        for(int i=1;i<=n;++i)
            if(!flag[i]){
                int delta=lh[v]+rh[i]-w[v][i];
                if(delta<slack[i])
                    slack[i]=delta,pre[i]=u;
                    //点i的匹配点有可能置换为u的匹配点,
                    //以腾出u的匹配点的空位
                if(minh>slack[i])
                    minh=slack[i],nxt=i;//点i下次将被访问
            }
        //松弛
        for(int i=0;i<=n;++i)
            if(flag[i])lh[pair[i]]-=minh,rh[i]+=minh;
            else slack[i]-=minh;
        u=nxt;
    } while(pair[u]);//直到找到未匹配点为止
    //置换匹配
    while(u) {
        int p=pre[u];
        pair[u]=pair[p];
        u=p;
    }
}
int KM(int n) {
    for(int i=1;i<=n;++i) {
        int maxh=0;
        for(int j=1;j<=n;++j)
            maxh=std::max(maxh,w[i][j]);
        lh[i]=maxh;
    }
    reset(rh);
    reset(pair);
    for(int i=1;i<=n;++i)
        aug(i);
    int res=0;
    for(int i=1;i<=n;++i)
        res+=w[pair[i]][i];
    return res;
}
```

实测该方法比优化 2 的效率更高(2x)。

2.1.4 二分图常见模型

2.1.4.1 最小点覆盖

定理 2.2 (König's Theorem) 最小点覆盖数 = 最大匹配数。

使用反证法证明: 如果有一条边两端顶点都不在最大匹配上, 那么这条边可以进入最大匹配成为一个更大的匹配边集, 所以与最大匹配的假设矛盾。

2.1.4.2 最大独立集

定理 2.3 最大独立集大小 = 顶点数-最小点覆盖数 = 顶点数-最大匹配数

证明：容易发现去掉二分图中的最小点覆盖可得到一个独立集(若其不是独立集，则说明存在一条边未被覆盖，与点覆盖的定义矛盾)。尝试以此独立集为基础扩展，可以发现若要使点覆盖中的某个点变为独立集的点，由最小点覆盖数 = 最大匹配数可知，最小点覆盖的每个点都与 ≥ 1 的边相连，因此必须使不少于 1 个原独立集的点被删除。所以无论如何修改，最多得到与之大小相等的独立集。

2.1.4.3 DAG 最小路径覆盖

最小不相交路径覆盖 将顶点拆成左右两点，若存在边 $u \to v$ 则连边 $Lu \to Rv$，求二分图最大匹配。

定理 2.4 最小路径覆盖数 = 顶点数-二分图最大匹配数。

证明：二分图中每增加一个匹配，就意味着减少一条路径。

最小可相交路径覆盖 先用 Floyd 求出传递闭包，转化为最小不相交路径覆盖问题。因为如果要从 a 走到 b，直接连边可以避开中间点的流量限制。

以上内容参考了罗茜[4]，justPassBy[5]和不可不戒[6] 的博客。

2.1.5 Hall 定理

Hall 定理用于判断二分图是否存在完美匹配。

定理 2.5 二分图 $G = \{V1, V2, E\}, |V1| \leq |V2|$ 存在完美匹配当且仅当 $V1$ 中任意 k 个顶点至少与 $V2$ 中任意 k 个顶点相连。

证明 充分性：假设二分图 G 不存在完美匹配，记 G 的最大匹配为 M，$V1$ 上至少有一点 u 不在 M 上。由条件可知点 u 有一条不在 M 上的边，记对面的点为 v。若点 v 不在 M 上，则与 M 为最大匹配矛盾；否则尝试使用匈牙利算法寻找增广路，记涉及到的 $V1$ 的子集为 S，则右边至少有 $|S|$ 个节点与其相连，因而存在增广路，与 M 为最大匹配矛盾。

必要性：由于二分图 G 有完美匹配，$V1$ 的 k 个顶点至少与各自的匹配相连。

还有一个比较有用的推论：

推论 2.6 对于二分图 $G = \{V1, V2, E\}, |V1| \leq |V2|$，若存在整数 t，满足 $V1$ 中任意节点的度数 $\geq t$，$V2$ 中任意节点的度数 $\leq t$，则 G 存在完美匹配。

例题 [POI2009]LYZ-Ice Skates

由定理 2.5可以考虑枚举所有集合，但复杂度无法接受，考虑排掉一些显然不优的集合。选出的集合可以分为 3 类：

- 脚的大小连续；
- 脚的大小不连续但是鞋号区间连续，把中间未被选中的脚的大小选中，但是鞋号区间不变，可以有更充分的证据证明不存在完美匹配；

[4]二分图详解及总结 https://www.cnblogs.com/alihenaixiao/p/4695298.html

[5]有向无环图(DAG)的最小路径覆盖 https://www.cnblogs.com/justPassBy/p/5369930.html

[6]二分图: 最大独立集 & 最大匹配 & 最小顶点覆盖 https://blog.csdn.net/lezg_bkbj/article/details/9872189

- 脚的大小不连续且鞋号区间不连续, 这个集合可以根据鞋号区间的连续性分为前两种集合, 每个集合是独立的子问题。

因此只需考虑脚的大小连续的集合。

记脚的大小为 i 的人数有 a_i 个, 根据定理有 $\sum_{i=l}^{r} a_i \leq (r+d-l+1)*k$。让右端为常数, 得 $\sum_{i=l}^{r} (a_i - k) \leq d*k$, 可用线段树维护最大子段和。

代码:

```
// P3488
#include <cstdio>
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
const int size = 500005 << 2;
typedef long long Int64;
Int64 maxv(Int64 a, Int64 b) {
    return a > b ? a : b;
}
Int64 L[size], R[size], sum[size], maxs[size];
void update(int id) {
    L[id] = maxv(L[id << 1],
                 sum[id << 1] + L[id << 1 | 1]);
    R[id] = maxv(R[id << 1] + sum[id << 1 | 1],
                 R[id << 1 | 1]);
    maxs[id] =
        maxv(maxv(maxs[id << 1], maxs[id << 1 | 1]),
             R[id << 1] + L[id << 1 | 1]);
}
#define ls l, m, id << 1
#define rs m + 1, r, id << 1 | 1
Int64 k;
void build(int l, int r, int id) {
    sum[id] = (l - r - 1) * k;
    if(l != r) {
        int m = (l + r) >> 1;
        build(ls);
        build(rs);
    }
```

```
}
int p, val;
void modify(int l, int r, int id) {
    sum[id] += val;
    if(l == r)
        L[id] = R[id] = maxs[id] = maxv(sum[id], 0);
    else {
        int m = (l + r) >> 1;
        if(p <= m)
            modify(ls);
        else
            modify(rs);
        update(id);
    }
}
int main() {
    int n = read();
    int m = read();
    k = read();
    Int64 key = read() * k;
    build(1, n, 1);
    while(m--) {
        p = read();
        val = read();
        modify(1, n, 1);
        puts(maxs[1] <= key ? "TAK" : "NIE");
    }
    return 0;
}
```

上述内容参考了 Feynman1999 的博客[7]。

2.2 最大流

Dinic 与 ISAP 属于 Ford-Fulkerson 方法中的 SAP(Shortest Augment Path)系。而 HLPP 属于 Push–Relabel 算法。

2.2.1 Dinic 算法

~~个人比较喜欢使用 Dinic 算法(因为我只会这个)~~。
ISAP 大法好!!!
Dinic 的计算流程如下:

1. BFS 建分层图, 若找不到增广路则退出;
2. DFS 在分层图上找增广路并修改流量, 重复步骤 1。

时间复杂度证明:

[7]Hall 定理（二分图匹配问题，Hungary 算法基础）https://blog.csdn.net/feynman1999/article/details/76037603

1. **引理 2.7** *Dinic* 每次 *BFS* 后的阻塞流层数是递增的(即 $d[t]$ 递增)。
2. 每次 BFS 的时间复杂度为 $O(E)$。
3. 每次 DFS 的时间复杂度为 $O(VE)$。

因此算法的时间复杂度为 $O(V^2E)$。
在容量均为 1 的图上, Dinic 的时间复杂度为 $O(min\{V^{\frac{2}{3}}, E^{\frac{1}{2}}\}E)$, 证明:
留坑待填, 参见 [1]。
做二分图最大匹配时 Dinic 跑得飞快, 时间复杂度 $O(\sqrt{V}E)$, 证明:
留坑待填, 参见 [2]。
时间复杂度证明源自 Wikipedia-EN[8]以及 permui 的博客[9]

Warning: 在特殊图中如果常规算法不能过, 考虑使用 Dinic 而不是目前常用的 ISAP, 可以获得一些玄学的加速(这是我第一次遇到 ISAP 一直返回 0 流), 或者使用贪心算法模拟网络流。

2.2.1.1 优化

- 当前弧优化: 每次从未遍历的边开始遍历, 减少重复计算(就算前面的边没满, 下一次还可以增广)。
- 记录无法增广的点(将其深度设为-1), 避免重复计算。
- (玄学, 未测试)BFS 找到一条增广路就退出, 无法解释。
- 若图为分层图, 在 Dinic 之前贪心预流(依旧玄学, 未测试):
  1. 从 s 开始逐层递推, 计算能够流出节点 i 的流量 $out[i]$;
  2. 从 t 开始逐层倒推, 计算每条边的实际流量。

代码:

PreFlow

```
int d[size], id[size], in[size], out[size];
bool cmp(int a, int b) {
    return d[a] < d[b];
}
void preFlow(int n, int s, int t) {
    for(int i = 1; i <= n; ++i)
        id[i] = i;
    std::sort(id + 1, id + n + 1, cmp);
    in[s] = inf;
    for(int i = 1; i <= n; ++i) {
        int u = id[i];
        for(int j = last[u]; j; j = E[j].nxt) {
            int v = E[j].to;
            if(d[v] == d[u] + 1) {
                int f =
                    std::min(in[u] - out[u], E[i].f);
                in[v] += f, out[u] += f;
```

[8]Dinic's algorithm - Wikipedia https://en.wikipedia.org/wiki/Dinic%27s_algorithm
[9]最大流算法-ISAP - permui https://www.cnblogs.com/owenyu/p/6852664.html

```
                }
            }
        }
        memset(in + 1, 0, sizeof(int) * n);
        in[f] = inf;
        for(int i = n; i >= 1; --i) {
            int u = id[i];
            for(int j = last[u]; j; j = E[j].nxt) {
                int v = E[j].to;
                if(d[v] + 1 == d[u]) {
                    int f = std::min(
                        std::min(out[v] - in[v], in[u]),
                        E[j ^ 1].f);
                    in[v] += f, in[u] -= f;
                    E[j].f += f, E[j ^ 1].f -= f;
                }
            }
        }
    }
```

该方法源自沐阳的博客。[10]

- 从终点开始建分层图, 避免花费过多时间经过非最短路。该建议来自大本营的博客[11]。该方法可沿用至 MCMF。

2.2.1.2 板子

常规优化:

DinicA

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 10005;
struct Edge {
    int to, nxt, f;
```

[10]ZOJ-2364 Data Transmission 分层图阻塞流 Dinic+ 贪心预流 - 沐阳 https://www.cnblogs.com/Lyush/p/3204099.html

[11]【2018.2.8-】网络流学习笔记(含 ISAP!)https://www.cnblogs.com/scx2015noip-as-php/p/MFP.html

```
} E[200005];
int last[size], cnt = 1;
void addEdgeImpl(int u, int v, int f) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].f = f;
    last[u] = cnt;
}
void addEdge(int u, int v, int f) {
    addEdgeImpl(u, v, f);
    addEdgeImpl(v, u, 0);
}
int q[size], d[size], S, T;
bool BFS(int siz) {
    memset(d + 1, -1, sizeof(int) * siz);
    d[S] = 0, q[0] = S;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(E[i].f && d[v] == -1) {
                d[v] = d[u] + 1;
                if(v == T)
                    return true;
                q[e++] = v;
            }
        }
    }
    return false;
}
int now[size];
int DFS(int u, int mf) {
    if(u == T || mf == 0)
        return mf;
    int res = 0, k;
    for(int& i = now[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(d[v] == d[u] + 1 &&
           (k = DFS(v, std::min(mf, E[i].f)))) {
            E[i].f -= k, E[i ^ 1].f += k;
            res += k, mf -= k;
            if(mf == 0)
                break;
        }
    }
    if(res == 0)
        d[u] = -1;
    return res;
}
int dinic(int siz) {
```

```
    int res = 0;
    while(BFS(siz)) {
        memcpy(now + 1, last + 1, sizeof(int) * siz);
        res += DFS(S, 1 << 30);
    }
    return res;
}
int main() {
    int n = read();
    int m = read();
    S = read();
    T = read();
    while(m--) {
        int u = read();
        int v = read();
        int w = read();
        addEdge(u, v, w);
    }
    printf("%d\n", dinic(n));
    return 0;
}
```

玄学优化(注意在随机数据下表现可能更差):

- 伸缩操作:首先按照边的容量从大到小排序,然后枚举位按照 $cap \geq 2^k, 2^{k-1}, \cdots, 2^0$ 加边,每加一组边跑一次 Dinic。时间复杂度 $O(VE \lg C)$。
- 延迟加反向边:建图时仍然加正反向边,但是第一次 Dinic 时避开反向边,第二次 Dinic 时才考虑反向边。
- 不退流跑,一次性退流:BFS 失败时才退流,若退流后仍然失败才退出迭代。

这些优化参见 kczno1 的博客[12]。
参考代码:
常规优化 + 伸缩操作 + 延迟加反向边(实践中还是这个比较好用):

DinicB

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
```

[12]论如何用 dinic ac 最大流加强版 http://kczno1.blog.uoj.ac/blog/3375

```
}
const int size = 1205, esiz = size * 100;
struct Edge {
    int to, nxt, f;
} E[esiz * 2];
int last[size], cnt = 0;
void addEdge(int u, int v, int f) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].f = f;
    last[u] = cnt;
}
int q[size], d[size], S, T, endPos;
bool BFS(int siz) {
    memset(d + 1, -1, sizeof(int) * siz);
    d[S] = 0, q[0] = S;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        for(int i = last[u]; i > endPos;
            i = E[i].nxt) {
            int v = E[i].to;
            if(E[i].f && d[v] == -1) {
                d[v] = d[u] + 1;
                if(v == T)
                    return true;
                q[e++] = v;
            }
        }
    }
    return false;
}
int now[size], neg[esiz * 2];
int DFS(int u, int mf) {
    if(u == T || mf == 0)
        return mf;
    int res = 0, k;
    for(int& i = now[u]; i > endPos; i = E[i].nxt) {
        int v = E[i].to;
        if(d[v] == d[u] + 1 &&
           (k = DFS(v, std::min(mf, E[i].f)))) {
            E[i].f -= k, E[neg[i]].f += k;
            res += k, mf -= k;
            if(mf == 0)
                break;
        }
    }
    if(res == 0)
        d[u] = -1;
    return res;
}
```

```
int res = 0;
void dinic(int siz) {
    while(BFS(siz)) {
        memcpy(now + 1, last + 1, sizeof(int) * siz);
        res += DFS(S, 2147483647);
    }
}
struct Info {
    int u, v, w;
    bool operator<(const Info& rhs) const {
        return w < rhs.w;
    }
} A[esiz];
int ep[2][35];
int main() {
    int n = read();
    int m = read();
    S = read();
    T = read();
    for(int i = 0; i < m; ++i) {
        A[i].u = read();
        A[i].v = read();
        A[i].w = read();
    }
    std::sort(A, A + m);
    for(int t = 1; t >= 0; --t) {
        for(int c = 0, i = 0; c <= 30; ++c) {
            ep[t][c] = cnt;
            while(i < m && (A[i].w >> (c + 1)) == 0) {
                if(t) {
                    addEdge(A[i].v, A[i].u, 0);
                    neg[cnt] = cnt + m;
                } else {
                    addEdge(A[i].u, A[i].v, A[i].w);
                    neg[cnt] = cnt - m;
                }
                ++i;
            }
        }
    }
    int lep = 1 << 30;
    for(int t = 0; t <= 1; ++t) {
        for(int c = 30; c >= 0; --c) {
            endPos = ep[t][c];
            if(lep != endPos) {
                dinic(n);
                lep = endPos;
            }
        }
    }
```

```
    printf("%d\n", res);
    return 0;
}
```

kczno1 的最新做法-不退流跑, 一次性退流:

DinicC

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1205;
struct Edge {
    int to, nxt, f, df;
} E[240005];
int last[size], cnt = 1;
void addEdgeImpl(int u, int v, int f) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].f = f;
    last[u] = cnt;
}
void addEdge(int u, int v, int f) {
    addEdgeImpl(u, v, f);
    addEdgeImpl(v, u, 0);
}
int q[size], d[size], S, T;
bool BFS(int siz) {
    memset(d + 1, -1, sizeof(int) * siz);
    d[S] = 0, q[0] = S;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(E[i].f && d[v] == -1) {
                d[v] = d[u] + 1;
                if(v == T)
                    return true;
                q[e++] = v;
            }
```

```
        }
    }
    return false;
}
int now[size];
int DFS(int u, int mf) {
    if(u == T || mf == 0)
        return mf;
    int res = 0, k;
    for(int& i = now[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(d[v] == d[u] + 1 &&
           (k = DFS(v, std::min(mf, E[i].f)))) {
            E[i].f -= k, E[i].df += k;
            res += k, mf -= k;
            if(mf == 0)
                break;
        }
    }
    if(res == 0)
        d[u] = -1;
    return res;
}
void cook() {
    for(int i = 2; i <= cnt; i += 2) {
        E[i].f += E[i ^ 1].df;
        E[i ^ 1].f += E[i].df;
        E[i].df = E[i ^ 1].df = 0;
    }
}
int dinic(int siz) {
    int res = 0;
    while(BFS(siz)) {
        do {
            memcpy(now + 1, last + 1,
                   sizeof(int) * siz);
            res += DFS(S, 2147483647);
        } while(BFS(siz));
        cook();
    }
    return res;
}
int main() {
    int n = read();
    int m = read();
    S = read();
    T = read();
    while(m--) {
        int u = read();
        int v = read();
```

```
        int w = read();
        addEdge(u, v, w);
    }
    printf("%d\n", dinic(n));
    return 0;
}
```

2.2.1.3 当 Dinic 遇上 LCT

留坑待补。

2.2.2 ISAP 算法

Dinic 每次 BFS 计算分层图的过程为找最短增广路的过程。每次 BFS 重新计算层次编号 d 似乎有些浪费, 因此 ISAP 在 Dinic 的基础上用 DFS 直接修改层次编号的方式来优化算法。ISAP 的时间复杂度仍然为 $O(V^2E)$。记数组 $d[u]$ 为残存网络中点 u 到汇点的最短距离, 为了编码方便让 $d[T] = 1$。

算法步骤如下:

- 迭代 DFS 增广, 若找不到满足 $d[u] = d[v] + 1$ 的可增广边则说明此时的最短路标号已经过时, 为了让点 u 可增广, 令 $d[u] = min\{d[v]\} + 1$。
- 若 $d[S] > |V|$ 则说明已不存在简单增广路径, 退出迭代。

2.2.2.1 优化

- 若数组 d 被初始化为 0, 则 DFS 需要 $O(n^2)$ 的时间来初始化数组 d。可以在增广前从汇点开始 BFS$O(n + m)$ 预处理数组 d。
- gap 优化: 维护每种层次编号的数量 $gap[d]$, 若 $gap[d] = 0$ 则说明出现了断层, 不存在新的增广路。此时简单地令 $d[S] = n + 1$ 结束算法。
- 类似 Dinic 可以使用当前弧优化, **但在层次标号被修改后要重置链头**。
- 层次标号的修改是连续的, 每次增广完后 $++d[u]$。
- 流量用完后直接退出。

板子(代码比 DinicA 还短而且跑得比 DinicB 还快):

ISAP

```
0 #include <algorithm>
  #include <cstdio>
  #include <cstring>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
      while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
10        c = getchar();
```

```
    }
    return res;
}
const int size = 10005;
struct Edge {
    int to, nxt, f;
} E[size * 20];
int last[size], cnt = 1;
void addEdgeImpl(int u, int v, int f) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].f = f;
    last[u] = cnt;
}
void addEdge(int u, int v, int f) {
    addEdgeImpl(u, v, f);
    addEdgeImpl(v, u, 0);
}
int d[size], q[size], gap[size], S, T, n;
void BFS() {
    q[0] = T, ++gap[1], d[T] = 1;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(!d[v]) {
                d[v] = d[u] + 1;
                ++gap[d[v]];
                q[e++] = v;
            }
        }
    }
    if(d[S] == 0)
        d[S] = 1 << 30;
}
int now[size];
int aug(int u, int mf) {
    if(u == T || mf == 0)
        return mf;
    int res = 0, k;
    for(int& i = now[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(d[u] == d[v] + 1 &&
           (k = aug(v, std::min(mf, E[i].f)))) {
            E[i].f -= k, E[i ^ 1].f += k;
            res += k, mf -= k;
            if(!mf)
                return res;
        }
    }
```

```
    if(--gap[d[u]] == 0)
        d[S] = n + 1;
    ++gap[++d[u]], now[u] = last[u];
    return res;
}
int ISAP() {
    BFS();
    memcpy(now + 1, last + 1, sizeof(int) * n);
    int res = 0;
    while(d[S] <= n)
        res += aug(S, 1 << 30);
    return res;
}
int main() {
    n = read();
    int m = read();
    S = read();
    T = read();
    while(m--) {
        int u = read();
        int v = read();
        int w = read();
        addEdge(u, v, w);
    }
    printf("%d\n", ISAP());
    return 0;
}
```

注意 $mf = 0$ 时直接返回不要更新层次标号。
为了防止初始 0 流的情况，需要特判并将 $d[S]$ 置为 $+\infty$。
ISAP 算法参考了 permui 的博客[13]。

2.2.3 HLPP 算法

~~算法导论~~ [4] ~~26.4 节讲的推送-重贴标签算法是~~ $O(V^3)$~~的~~。。。
HLPP 算法使用“推送-重贴标签”算法，其时间复杂度为 $O(V^2\sqrt{E})$。虽然时间复杂度比 Dinic 优，但由于 HLPP 算法上界较紧，在实践中往往跑不过 Dinic（加了优化后表现还行）。

2.2.3.1 推送-重贴标签算法

以水流类比网络流，每条边都是一根有流量限制的水管，允许每个点暂时存储一些多余的水，称为超额流。特别地，源汇点可以长期存储无限多的水。其它点需要伺机将自身的超额流推送出去，这里给每个节点再引入一个“高度”参数，规定流量只能往低处走。固定源点的高度为 V。当某个节点高于源点时，它的超额流将退回给源点。**注意高度可以达到** $2V - 1$
该算法由两个基本操作组成：

- “推送”：一个节点把自己的超额流推送给高度比自己低 1 的节点（源点无高度差限制）。

[13]最大流算法-ISAP - permui https://www.cnblogs.com/owenyu/p/6852664.html

- “重贴标签”: 当一个节点无法推送完超额流时, 将自身高度加到连边有残存流量的最低邻接点的高度 $+1$。

首先令 S 的出边满流, 然后维护超额流节点队列, 每次取出节点对其进行推送或重贴标签操作。直至不存在超额流节点。时间复杂度 $O(V^2E)$。

2.2.3.2 前置重贴标签算法

每次重贴标签时将节点移至队首, 可将时间复杂度优化至 $O(V^3)$。
参见算法导论 [4] 第 26.5 节。

2.2.3.3 HLPP 实现与优化

使用优先队列以高度为关键字维护超额流节点, 每次选取最高标号的节点进行 “推送-重贴标签”。
优化:

- gap 优化: 当一个点被重贴标签后, 若没有其他点拥有其原来的高度, 高于此高度的点就无法把流量推送到汇点。将这些点的高度全部设为 $V+1$ 使其流量流回源点。
- 高度预计算(我因此而 TLE 多次): 将 d 初始化为每个点到汇点的最短路径长。**注意源点的高度固定为 V。**
- 使用桶维护优先队列: 注意到高度值的范围不大, 使用桶来维护较为快速。

板子:
优先队列版:

HLPPA

```
// 4722
#include <cstdio>
#include <cstring>
#include <queue>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1205;
struct Edge {
    int to, nxt, w;
} E[240005];
int last[size], cnt = 1;
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].w = w;
```

```
    last[u] = cnt;
}
int f[size], gap[size << 1], h[size];
struct Cmp {
    bool operator()(int a, int b) const {
        return h[a] < h[b];
    }
};
int push(int u, int v, int id, int fu) {
    int w = std::min(fu, E[id].w);
    f[u] -= w, f[v] += w, E[id].w -= w,
        E[id ^ 1].w += w;
    printf("push %d %d %d\n", u, v, w);
    return w;
}
int q[size];
bool flag[size];
void pre(int n, int t) {
    memset(h, 0x3f, sizeof(h));
    h[t] = 0, flag[t] = true, q[0] = t;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        if(b == size)
            b = 0;
        flag[u] = false;
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(E[i ^ 1].w && h[v] > h[u] + 1) {
                h[v] = h[u] + 1;
                q[e++] = v;
                if(e == size)
                    e = 0;
            }
        }
    }
}
const int inf = 2147483647;
int solve(int n, int s, int t) {
    std::priority_queue<int, std::vector<int>, Cmp>
        heap;
    pre(n, t);
    if(h[s] == 0x3f3f3f3f)
        return 0;
    h[s] = n;
    for(int i = 1; i <= n; ++i)
        if(h[i] != 0x3f3f3f3f)
            ++gap[h[i]];
    flag[s] = flag[t] = true;
    for(int i = last[s]; i; i = E[i].nxt) {
```

```
        int v = E[i].to;
        if(push(s, v, i, inf) && !flag[v])
            heap.push(v), flag[v] = true;
    }
    while(heap.size()) {
        int u = heap.top();
        int hu = h[u];
        heap.pop();
        flag[u] = false;
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(hu == h[v] + 1 && push(u, v, i, f[u]) &&
               !flag[v]) {
                heap.push(v), flag[v] = true;
                if(!f[u])
                    break;
            }
        }
        if(f[u]) {
            if(--gap[hu] == 0) {
                for(int i = 1; i <= n; ++i)
                    if(i != s && i != t && hu < h[i] &&
                       h[i] <= n)
                        h[i] = n + 1;
            }
            h[u] = inf;
            for(int i = last[u]; i; i = E[i].nxt)
                if(E[i].w)
                    h[u] =
                        std::min(h[u], h[E[i].to] + 1);
            ++gap[h[u]];
            heap.push(u);
            flag[u] = true;
        }
    }
    return f[t];
}
int main() {
    int n = read();
    int m = read();
    int s = read();
    int t = read();
    while(m--) {
        int u = read();
        int v = read();
        int w = read();
        addEdge(u, v, w);
        addEdge(v, u, 0);
    }
    printf("%d\n", solve(n, s, t));
```

```
    return 0;
}
```

桶版(参考 PM250 的代码[14], 自己不会用 vector 然后就用 set 代替了, 常数大好多):

HLPPB

```
#include <cstdio>
#include <cstring>
#include <set>
#include <vector>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1205;
struct Edge {
    int to, nxt, w;
} E[240005];
int last[size], cnt = 1;
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].w = w;
    last[u] = cnt;
}
const int inf = 2147483647;
std::set<int> dl[size << 1];
std::vector<int> heap[size << 1];
typedef std::multiset<int>::iterator Iter;
int highest = 0, highestInHeap = -1, h[size], f[size],
    q[size];
bool flag[size];
void pre(int n, int t) {
    memset(h, 0x3f, sizeof(h));
    h[t] = 0, flag[t] = true, q[0] = t;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        if(b == size)
            b = 0;
        flag[u] = false;
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
```

[14]R13845988 评测详情 https://www.luogu.org/record/show?rid=13845988

```
            if(E[i ^ 1].w && h[v] > h[u] + 1) {
                h[v] = h[u] + 1;
                q[e++] = v;
                if(e == size)
                    e = 0;
            }
        }
    }
}
int push(int u, int v, int id, int fu) {
    int w = std::min(fu, E[id].w);
    f[u] -= w, f[v] += w, E[id].w -= w,
        E[id ^ 1].w += w;
    return w;
}
int HLPP(int n, int s, int t) {
    f[s] = inf;
    pre(n, t);
    if(h[s] == 0x3f3f3f3f)
        return 0;
    h[s] = n;
    for(int i = 1; i <= n; ++i)
        if(i != s && h[i] != 0x3f3f3f3f) {
            dl[h[i]].insert(i);
            highest = std::max(highest, h[i]);
        }
    flag[s] = flag[t] = true;
    for(int i = last[s]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(push(s, v, i, inf) && !flag[v]) {
            heap[h[v]].push_back(v);
            flag[v] = true;
            highestInHeap =
                std::max(highestInHeap, h[v]);
        }
    }
    while(highestInHeap >= 0) {
        if(heap[highestInHeap].size()) {
            int u = heap[highestInHeap].back();
            heap[highestInHeap].pop_back();
            flag[u] = false;
            int minh = inf;
            for(int i = last[u]; i && f[u];
                i = E[i].nxt) {
                int v = E[i].to;
                if(h[u] == h[v] + 1) {
                    if(push(u, v, i, f[u]) &&
                       !flag[v]) {
                        heap[h[v]].push_back(v);
                        flag[v] = true;
```

```
                        highestInHeap = std::max(
                            highestInHeap, h[v]);
                    }
                } else if(E[i].w)
                    minh = std::min(minh, h[v] + 1);
            }
            if(f[u]) {
                if(dl[highestInHeap].size() == 1) {
                    for(int i = highestInHeap;
                        i <= highest; ++i) {
                        for(Iter it = dl[i].begin();
                            it != dl[i].end(); ++it)
                            h[*it] = n + 1;
                        dl[highestInHeap].clear();
                    }
                    highest = highestInHeap - 1;
                } else
                    dl[highestInHeap].erase(u);

                h[u] = minh;
                heap[minh].push_back(u);
                dl[minh].insert(u);
                flag[u] = true;
                highestInHeap =
                    std::max(highestInHeap, minh);
                highest = std::max(highest, minh);
            }
        } else
            --highestInHeap;
    }
    return f[t];
}
int main() {
    int n = read();
    int m = read();
    int s = read();
    int t = read();
    while(m--) {
        int u = read();
        int v = read();
        int w = read();
        addEdge(u, v, w);
        addEdge(v, u, 0);
    }
    printf("%d\n", HLPP(n, s, t));
    return 0;
}
```

HLPP 算法参考了 Mr_Spade 的博客[15]。

[15]网络最大流——最高标号预流推进 https://www.cnblogs.com/Mr-Spade/p/9636935.html

2.2.4 最大流与最小割

定理 2.8 (Max-flow min-cut theorem) 最大流 = 最小割。

证明:

- **引理 2.9** 最大流 ≤ 最小割
  由于流量被割边所限制, 所以最大流 ≤ 任意割, 所以最大流 ≤ 最小割。
- **引理 2.10** 最大流 ≥ 最小割
  证明: 跑完最大流后残量网络内 s 与 t 不连通, 所以得到了一个割, 即最大流 ≥ 最小割。

结合引理 2.9与 2.10可得最大流 = 最小割。

2.2.5 无向图最小割

2.2.5.1 Stoer-Wagner Algorithm

若需要求全局最小割, 使用 Stoer-Wagner Algorithm。
算法步骤如下:

1. 任意指定一个节点作为初始点集;
2. 查询 [到 [点集内的点] 边权和最大] 的 [点集外的点];
3. 合并最后加入的两个节点 s, t 并更新最小割;
4. 重复第一步直至整个图被合并。

具体做法见代码。边权可用优先队列维护, 时间复杂度 $O(|V||E|\lg|E|)$。
模板(SP12056 FZ10B - Nubulsa Expo):

```
// SP12056
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <queue>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 305;
struct Edge {
    int to, nxt, w;
} E[100005];
int last[size], cnt;
```

```
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].w = w;
    last[u] = cnt;
}
int fa[size], nxt[size];
template <typename T>
void reset(T* arr, int siz) {
    memset(arr + 1, 0, sizeof(T) * siz);
}
void init(int n) {
    cnt = 0;
    reset(last, n);
    reset(fa, n);
    reset(nxt, n);
}
int find(int x) {
    return fa[x] ? fa[x] = find(fa[x]) : x;
}
void merge(int u, int v) {
    int p = u;
    while(nxt[p])
        p = nxt[p];
    nxt[p] = v;
    fa[v] = u;
}
int w[size];
bool flag[size];
struct Info {
    int id, f;
    Info(int id) : id(id), f(w[id]) {}
    bool operator<(const Info& rhs) const {
        return f < rhs.f;
    }
};
int solveImpl(int n, int cnt, int& s, int& t) {
    reset(w, n);
    reset(flag, n);
    t = 1;
    std::priority_queue<Info> heap;
    while(cnt--) {
        s = t;
        flag[s] = true;
        for(int i = s; i; i = nxt[i]) {
            for(int j = last[i]; j; j = E[j].nxt) {
                int v = find(E[j].to);
                if(!flag[v]) {
                    w[v] += E[j].w;
                    heap.push(v);
                }
```

```
            }
        }
        t = 0;
        while(!t) {
            if(heap.empty())
                return 0;
            Info info = heap.top();
            heap.pop();
            if(w[info.id] == info.f)
                t = info.id;
        }
    }
    merge(s, t);
    return w[t];
}
int solve(int n) {
    int res = 1 << 30, s, t;
    for(int i = n - 1; i && res; --i)
        res = std::min(res, solveImpl(n, i, s, t));
    return res;
}
int main() {
    while(true) {
        int n = read();
        int m = read();
        read();
        if(n == 0)
            break;
        init(n);
        while(m--) {
            int u = read();
            int v = read();
            int w = read();
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        printf("%d\n", solve(n));
    }
    return 0;
}
```

这题 $|V|$ 比较小所以可以用邻接矩阵存图, $O(|V|^3)$ 解决。

```
// SP12056
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
```

```
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 305;
typedef long long Int64;
Int64 A[size][size];
bool flag[size], vis[size];
void merge(int u, int v, int n) {
    for(int i = 1; i <= n; ++i) {
        A[i][u] += A[i][v];
        A[u][i] += A[v][i];
    }
    flag[v] = true;
}
Int64 w[size];
Int64 solveImpl(int n, int cnt, int& s, int& t) {
    memcpy(vis + 1, flag + 1, sizeof(bool) * n);
    memset(w + 1, 0, sizeof(Int64) * n);
    for(int i = 1; i <= n; ++i)
        if(!flag[i]) {
            t = i;
            break;
        }
    while(cnt--) {
        s = t;
        vis[s] = true;
        t = 0;
        for(int i = 1; i <= n; ++i)
            if(!vis[i]) {
                w[i] += A[s][i];
                if(w[i] >= w[t])
                    t = i;
            }
    }
    merge(s, t, n);
    return w[t];
}
Int64 solve(int n) {
    memset(flag + 1, 0, sizeof(bool) * n);
    Int64 res = 1LL << 60;
    int s, t;
    for(int i = n - 1; i && res; --i)
        res = std::min(res, solveImpl(n, i, s, t));
    return res;
}
int main() {
```

```
    while(true) {
        int n = read();
        int m = read();
        read();
        if(n == 0)
            break;
        memset(A, 0, sizeof(A));
        while(m--) {
            int u = read();
            int v = read();
            int w = read();
            A[u][v] += w;
            A[v][u] += w;
        }
        printf("%lld\n", solve(n));
    }
    return 0;
}
```

不知为何两种方法在 SPOJ 上都 TLE 了。上述内容参考了 Oyking 的博客[16]。

2.2.5.2 流量构造法

若指定源汇点, 连边时给正反向边的残余流量都初始化为割边代价, 然后跑 Dinic。

2.2.6 最小割性质

该性质源自 AHOI2009 最小割, 描述了边在最小割中的条件。
首先求出任意一个 ST 割, 然后对残量网络求 SCC。
按照下列定理判定:

- 若该边满流且两端不属于同一个 SCC, 则该边出现在某个最小割中
- 若该边满流且起点与 S 在同一个 SCC 中, 终点与 T 在同一个 SCC 中, 则该边出现在任意最小割中

2.3 费用流

从普通的 EK 算法扩展, 既然每次增加的流量是一样的, 那么我们就选择费用最小(大)的增广路径, 从而保证在得到最大流的前提下费用最小(大)。
求最短路时使用 SPFA, 若没有负权边尽量使用 Dijkstra。
一般的建图思路是通过流量限制来保证方案合法, 然后设计边的费用引导至最优代价。

2.3.1 使用 Dijkstra 实现费用流

其实即使有负权边, 也是可以使用 Dijkstra 来求费用流的 ~~(但是仍然需要 SPFA)~~。
核心思想是对原图适当地修改变为不带负权边的图。首先在迭代外用 SPFA 求从源点到每个点的最短路, 记距离为 $h[u]$, 满足三角不等式 $h[u] + w[u][v] \geq h[v]$。将该式变形

[16]全局最小割 StoerWagner 算法详解 https://www.cnblogs.com/oyking/p/7339153.html

得 $w[u][v]+h[u]-h[v]\geq 0$, 令左式为边的新权值, 就可以在迭代中使用 Dijkstra 求最短路了, 记实际最短距离为 $md[i]$, 计算得到的最短距离为 $dis[i]$。容易发现最短路径边权和中的 $h[]$ 抵消后, 可以得到 $dis[i]=md[i]+h[S]-h[i]$, 其中 $h[S]=0$, 那么实际距离比计算距离多 $h[i]$。由此可得本次迭代产生的费用贡献为 $(dis[T]+h[T])*minf$, 并且需要在当前迭代结束前更新最短距离 $h'[u]=h[u]+dis[u]$。

上述内容参考了 Mogician 的博客[17]。

2.3.2 多路增广费用流

一般使用该方法作为费用流模板。
与普通费用流的差别如下:

- 使用 vector 存边对缓存友好
- SPFA 使用 SLF 带容错优化
- SPFA 从 T 开始找增广路
- DFS 多路增广从 S 沿着最短路跑, 可以使用当前弧优化, 注意不要走环

参考代码:

```
#include <cstdio>
#include <cstring>
#include <vector>
const int size = 405, inf = 2147483647;
struct Edge {
    int v, rev, f, w;
    Edge(int v, int rev, int f, int w)
        : v(v), rev(rev), f(f), w(w) {}
};
std::vector<Edge> G[size];
void addEdge(int u, int v, int f, int w) {
    int urev = G[u].size(), vrev = G[v].size();
    G[u].emplace_back(v, vrev, f, w);
    G[v].emplace_back(u, urev, 0, -w);
}
int dis[size], q[size], n;
bool flag[size];
bool SPFA() {
    memset(dis + 1, 0x3f, sizeof(int) * n);
    dis[n] = 0;
    int b = 0, e = 1;
    q[0] = n, flag[n] = true;
    while(b != e) {
        int u = q[b++];
        if(b == size)
            b = 0;
        flag[u] = false;
        for(auto& E : G[u]) {
```

[17]最大流与 Dijkstra 做费用流 - Mogician's blog - 洛谷博客 https://www.luogu.org/blog/Mogician/Network-Flow-Guide

```
            int v = E.v, cd = dis[u] - E.w;
            if(G[v][E.rev].f && cd < dis[v]) {
                dis[v] = cd;
                if(!flag[v]) {
                    flag[v] = true;
                    if(b == e ||
                       dis[q[b]] + 30 <= dis[v]) {
                        q[e++] = v;
                        if(e == size)
                            e = 0;
                    } else {
                        --b;
                        if(b == -1)
                            b = size - 1;
                        q[b] = v;
                    }
                }
            }
        }
    }
    return dis[1] != 0x3f3f3f3f;
}
int now[size];
int mini(int a, int b) {
    return a < b ? a : b;
}
int DFS(int u, int f) {
    if(u == n || f == 0)
        return f;
    flag[u] = true;
    int res = 0, k;
    for(int& i = now[u]; i < G[u].size(); ++i) {
        Edge& E = G[u][i];
        int v = E.v;
        if(E.f && !flag[v] && dis[v] + E.w == dis[u] &&
           (k = DFS(v, mini(f, E.f)))) {
            E.f -= k, G[v][E.rev].f += k;
            f -= k, res += k;
            if(f == 0)
                break;
        }
    }
    flag[u] = false;
    if(!res)
        dis[u] = -1;
    return res;
}
int main() {
    int m;
    scanf("%d%d", &n, &m);
```

```
    for(int i = 1; i <= m; ++i) {
        int u, v, f, w;
        scanf("%d%d%d%d", &u, &v, &f, &w);
        addEdge(u, v, f, w);
    }
    int maxF = 0, minW = 0;
    while(SPFA()) {
        memset(now + 1, 0, sizeof(int) * n);
        int f, sf = 0, cd = dis[1];
        while((f = DFS(1, inf)))
            sf += f;
        maxF += sf, minW += sf * cd;
    }
    printf("%d %d\n", maxF, minW);
    return 0;
}
```

该方法来自 Melacau 的博客(翻 fjsdfzoj 时发现的)[18]。

2.3.3 消圈定理

该定理用来求给定流量的最小费用流。

定理 2.11 (消圈定理) 当前流为给定流量的最小费用流当且仅当其残量网络中没有负环。

寻找负环可以使用 IDDFS-SPFA 解决。
该内容参考了 Sengxian 的博客[19]。

2.3.4 zkw 费用流

发现 zkw 费用流的实现比较短, 在性能要求不高时使用该方法作为费用流模板。zkw 费用流一般适用于流量大, 费用范围小, 增广路径短的图。

zkw 费用流同样使用多路增广思想, 与上文的不同之处在于其最短路算法是连续执行的。正确性证明留坑待补。

参考代码:

```
#include <cstdio>
int mini(int a, int b) {
    return a < b ? a : b;
}
const int size = 405, inf = 2147483647;
struct Edge {
    int to, nxt, f, w;
} E[30005];
int last[size], cnt = 1;
void addEdgeImpl(int u, int v, int f, int w) {
    ++cnt;
```

[18]【模板】板子的集合
https://www.cnblogs.com/Melacau/p/ban.html
[19]网络流之 - 消圈定理(POJ 2175)
https://blog.sengxian.com/algorithms/clearcircle

```
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].f = f,
    E[cnt].w = w;
    last[u] = cnt;
}
void addEdge(int u, int v, int f, int w) {
    addEdgeImpl(u, v, f, w);
    addEdgeImpl(v, u, 0, -w);
}
int n, dis[size], maxF = 0, minW = 0, vis[size],
                  ts = 0;
bool update() {
    int mdel = inf;
    for(int u = 1; u <= n; ++u) {
        if(vis[u] != ts)
            continue;
        for(int i = last[u]; i; i = E[i].nxt)
            if(E[i].f && vis[E[i].to] != ts)
                mdel = mini(mdel, dis[E[i].to] +
                                  E[i].w - dis[u]);
    }
    if(mdel == inf)
        return false;
    for(int u = 1; u <= n; ++u)
        if(vis[u] == ts)
            dis[u] += mdel;
    return true;
}
int aug(int u, int f) {
    if(u == n || f == 0) {
        maxF += f, minW += f * dis[1];
        return f;
    }
    int res = 0;
    vis[u] = ts;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(vis[v] != ts && E[i].f &&
           dis[v] == dis[u] - E[i].w) {
            int k = aug(v, mini(f, E[i].f));
            E[i].f -= k, E[i ^ 1].f += k;
            f -= k, res += k;
            if(f == 0)
                break;
        }
    }
    return res;
}
int main() {
    int m;
    scanf("%d%d", &n, &m);
```

```
    for(int i = 1; i <= m; ++i) {
        int u, v, f, w;
        scanf("%d%d%d%d", &u, &v, &f, &w);
        addEdge(u, v, f, w);
    }
    do {
        do
            ++ts;
        while(aug(1, inf));
    } while(update());
    printf("%d %d\n", maxF, minW);
    return 0;
}
```

该内容参考了 Angel_Kitty[20]和小蒟蒻 yyb[21]的博客。

2.4 带上下界网络流

前置知识:最大流与费用流。

2.4.1 无源汇有上下界可行流

最大流只能求带上界的网络流, 考虑通过修改建图方式使边的流量下界为 0, 流量上界对应变为上界-下界。若在修改后的图中, 边 e 使流量从点 u 流出 f 个单位, 则实际上边 e 通过 $f+low_e$ 单位的流量, 这时需要额外让点 u 流出 low_e 单位的流量, 同理流入流量时也需要额外接收 low_e 单位的流量。考虑设立虚拟节点 S,T 与顶点连附加边额外补偿/接收顶点的额外流量, 最后做 $S-T$ 的最大流使附加边满流。若附加边满流则说明可以满足每条边的流量下界限制, 存在可行解。此时每条边的实际流量即为图中流量 + 流量下界。建图:对于一条边 (u,v,low,up), 连边 $(S,v,low),(u,T,low),(u,v,up-low)$。

2.4.1.1 建图优化

普通方法建图需要连 $3E$ 条边, 但我们发现如果顶点 u 同时连接 S,T, 两条附加边的流量可以抵消掉一部分。因此可以在建图时预处理数组 $d[i]$, 维护 S 到 v 的流量与 v 到 T 的流量之差, 最后 $O(n)$ 选择 S 或 T 连边。使用此方法可以将边数降到 $V+E$。

2.4.1.2 费用流

如果按照常规方法连边, 由于附加边的费用不相等, 不能使用上述优化合并附加边。由于当且仅当附加边满流时才存在可行解, 可以先算上附加边的费用, 连附加边时把费用当成 0, 就可以继续使用建图优化了。

2.4.2 有上下界带源汇点可行流

若点 u 需要凭空生成流量 f, 从 S 向 u 连流量为 f 的边;若点 u 需要凭空消灭流量 f, 从 u 向 T 连流量为 f 的边;其余做法与子节 2.4.1相同。

[20]详解 zkw 算法解决最小费用流问题 https://www.cnblogs.com/ECJTUACM-873284962/p/7744943.html
[21]【LOJ#3097】[SNOI2019] 通信(费用流)https://www.cnblogs.com/cjyyb/p/10790579.html

2.4.3 有上下界带一组无限流量源汇点可行流

易知 S 的流出等于 T 的流入，因此加边 (T, S, inf) 后做法与子节 2.4.1相同。最终该边的流量即为 S 到 T 的总流量。

2.4.3.1 最大流

1. 先求出可行流；
2. 在残量网络上跑 $S \to T$ 的最大流；
3. 答案即为**最终残量网络**中 $T \to S$ 的流量 + 最大流流量。

正确性证明：

性质 2.12 求解最大流时不会修改源汇点路径之外的边的流量。

因此步骤 2 仍然能够保证该流是一个可行流(附加边仍然满流)。

2.4.3.2 最小流

1. 先求出可行流；
2. 在残量网络上跑 $T \to S$ 的最大流；
3. 答案即为**最终残量网络**中 $T \to S$ 的流量-最大流流量。$T \to S$ 的最大流相当于将 $S \to T$ 的流量减至最小。**注意得到答案为负的情况，此时应该令答案为 0。**

利用性质 2.12我们仍然能保证该流是一个可行流。
上述内容参考了 F.W.Nietzsche[22]与 liu_runda[23]的博客。

2.5 常见网络流/最小割模型

2.5.1 平面图转对偶图

平面图与对偶图的定义：

- 平面图(Planar Graph)：在平面上画出来可以使边与边只在顶点上相交的图。
- 对偶图(Dual Graph)：将平面图的每条边两边的区域连边而成的新平面图。

记平面图 G 的对偶图为 G^*，平面集合为 P_G。
对偶图 G^* 有两个性质：

- **性质 2.13** G^* 中的环对应 G 中的一个割。
- **性质 2.14** $|P_G| = |V_{G^*}|, |E_G| = |E_{G^*}|$

实际应用时，首先连接 (s, t) 使得外部平面被分为两个平面，以获得源汇点 s', t'(同时连到一个点上并没有什么用)，然后按照定义建图(不要连 s' 与 t' 的边)。
那么 s, t 的最小割 $= s' \to t'$ 的最短路 (即拆点前的最小环)，时间复杂度优于常规做法。

[22]有上下界的、有多组源汇的、网络流、费用流问题 - F.W.Nietzsche https://www.cnblogs.com/nietzsche-oier/p/8185805.html
[23]有上下界的网络流学习笔记 - liu_runda https://www.cnblogs.com/liu-runda/p/6262832.html

2.5.1.1 例题

Luogu P4001 [BJOI2006] 狼抓兔子[24]

根据定理 2.8转换为求最大流, 将右上角当做起点, 右下角当做终点, 然后使用上述方法转为对偶图求最短路。

Luogu P4001

```
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1010, vsiz = size * size * 2;
struct Edge {
    int to, next, w;
} E[3 * vsiz];
int last[vsiz] = {}, cnt = 0;
void addEdgeImpl(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].next = last[u], E[cnt].w = w;
    last[u] = cnt;
}
void addEdge(int u, int v, int w) {
    addEdgeImpl(u, v, w);
    addEdgeImpl(v, u, w);
}
const int S = 1, T = 2;
int dis[vsiz], q[vsiz];
bool flag[vsiz] = {};
void SPFA() {
    memset(dis, 0x3f, sizeof(dis));
    int b = 0, e = 1;
    q[0] = S, dis[S] = 0, flag[S] = true;
    while(b != e) {
        int u = q[b];
        flag[u] = false;
        b = (b + 1) % vsiz;
        for(int i = last[u]; i; i = E[i].next) {
            int v = E[i].to;
            if(dis[v] > dis[u] + E[i].w) {
                dis[v] = dis[u] + E[i].w;
```

[24]【P4001】[BJOI2006] 狼抓兔子 - 洛谷 https://www.luogu.org/problemnew/show/P4001

```
                if(!flag[v]) {
                    flag[v] = true;
                    q[e] = v;
                    e = (e + 1) % vsiz;
                }
            }
        }
    }
}
int ID[size][size][2];
int main() {
    int n = read();
    int m = read();
    int icnt = 2;
    for(int i = 1; i < n; ++i)
        for(int j = 1; j < m; ++j) {
            ID[i][j][0] = ++icnt;
            ID[i][j][1] = ++icnt;
        }
    for(int i = 1; i <= n; ++i)
        ID[i][0][0] = ID[i][0][1] = S,
        ID[i][m][0] = ID[i][m][1] = T;
    for(int i = 1; i <= m; ++i)
        ID[0][i][0] = ID[0][i][1] = T,
        ID[n][i][0] = ID[n][i][1] = S;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j < m; ++j)
            addEdge(ID[i - 1][j][0], ID[i][j][1],
                    read());
    for(int i = 1; i < n; ++i)
        for(int j = 1; j <= m; ++j)
            addEdge(ID[i][j - 1][1], ID[i][j][0],
                    read());
    for(int i = 1; i < n; ++i)
        for(int j = 1; j < m; ++j)
            addEdge(ID[i][j][0], ID[i][j][1], read());
    SPFA();
    printf("%d\n", dis[T]);
    return 0;
}
```

2.5.2 最大权闭合子图

关键字——依赖

S 向非负权点连容量为权值的边, 负权点向 T 连容量为权值相反数的边, 如果选择点 u 必须选择点 v, 就从 u 向 v 连容量为 ∞ 的边。

答案 $=$ 正权值之和-最小割。

简单理解: 如果割去正权点的权值, 则说明舍弃该正权点, 权值从答案中扣除; 如果割去负权点的权值, 则说明选择之前的正权点并从答案扣除该负权值。

严格的正确性证明待补充。

2.5.2.1 板子

Luogu P4174 [NOI2006] 最大获利[25]

Luogu P4174

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 55005, S = 0;
int T;
struct Edge {
    int to, nxt, f;
} E[6 * size];
int last[size] = {}, cnt = 1;
void addEdgeImpl(int u, int v, int f) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].f = f;
    last[u] = cnt;
}
void addEdge(int u, int v, int f) {
    addEdgeImpl(u, v, f);
    addEdgeImpl(v, u, 0);
}
const int inf = 1 << 30;
int d[size], q[size];
bool BFS(int siz) {
    memset(d, 0, sizeof(int) * siz);
    int b = 0, e = 1;
    q[0] = S;
    d[S] = 1;
    while(b != e) {
        int u = q[b++];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(E[i].f && d[v] == 0) {
                d[v] = d[u] + 1;
```

[25]【P4174】[NOI2006] 最大获利 - 洛谷 https://www.luogu.org/problemnew/show/P4174

```
                q[e++] = v;
            }
        }
    }
    return d[T];
}
int now[size];
int DFS(int u, int f) {
    if(u == T || f == 0)
        return f;
    int res = 0, k;
    for(int& i = now[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(d[v] == d[u] + 1 &&
           (k = DFS(v, std::min(E[i].f, f)))) {
            E[i].f -= k;
            E[i ^ 1].f += k;
            f -= k;
            res += k;
            if(f == 0)
                break;
        }
    }
    if(res == 0)
        d[u] = -1;
    return res;
}
int dinic(int siz) {
    int res = 0;
    while(BFS(siz)) {
        memcpy(now, last, sizeof(int) * siz);
        res += DFS(S, inf);
    }
    return res;
}
int main() {
    int n = read();
    int m = read();
    T = n + m + 1;
    for(int i = 1; i <= n; ++i)
        addEdge(i, T, read());
    int tot = 0;
    for(int i = 1; i <= m; ++i) {
        int a = read();
        int b = read();
        int c = read();
        tot += c;
        addEdge(S, i + n, c);
        addEdge(i + n, a, inf);
        addEdge(i + n, b, inf);
```

```
    }
    printf("%d\n", tot - dinic(T + 1));
    return 0;
}
```

2.5.2.2 输出方案

定理 2.15 *Dinic* 最后一次增广时可访问到的点的集合就是最优方案。

简单理解: 最后一次增广后 BFS 必然找不到增广路。若割源点到正权点的边, 则访问不到该正权点, 说明该正权点被舍弃, 若不被割则必割后继负权点对应的边, 该点可被访问到, 说明该正权点被选中; 若割负权点到源点的边, 则可访问到该负权点, 说明该负权点被选中, 若不被割则必割前驱正权点对应的边, 该点不可访问, 说明该点被舍弃。

上述内容参考了 appgle[26], MaxMercer[27]和 Cold_Chair[28] 的博客。

2.5.2.3 最大密度子图

给定一个无向图, 点有点权, 边有边权, 选择一个子图, 最大化边权和与点权和的比值。

由比值可以想到分数规划问题, 考虑二分密度 x, 满足 $\frac{Sum_E}{Sum_V} > x$, 继而推出 $Sum_E - Sum_V x > 0$, 不能取等号是因为不能选择空集。

由于选择一条边必定选择它的两个端点, 且恰好边有价值, 点有代价, 可以将其转化为最大权闭合子图问题求出 $Max\{Sum_E - Sum_V x\}$。因为容易求得选择方案, Dinkelbach 法也是不错的选择。

2.5.3 区间 k 覆盖问题

每个区间有价值, 选择一个区间集合, 满足每个位置最多被 k 个区间覆盖, 最大化价值和。

该问题可以转化为求 k 个不相交区间集合, 最大化价值和。可以这样建图: 区间上的点从左到右连边 $(i, i+1, \infty, 0)$, 每个区间连边 $(l, r+1, 1, w)$, S 连向左端点 $(k, 0)$, T 就是点 $n+1$。S 的流量限制保证了集合个数最多为 k, 每个区间的连边保证了同一条增广路内选择的区间不相交。

如果某些区间左端点相同且互斥, 则从左端点拆出一条容量为 1 的边限制选择。

2.5.4 集合划分问题

例题: POJ3469 Dual Core CPU

将 n 个元素划分为 2 个集合, 划分到 A 集合代价为 a_i, 划分到 B 集合代价为 b_i, 特别地还有 m 个二元组 (x_i, y_i), 若元素 x_i 与 y_i 不在同一个集合内, 要额外付出代价 c_i。求最小代价。

首先考虑没有额外代价如何建图。为了求最小代价, 要往最小割的方向思考。那么可以给每个元素建点, 从 S 向 i 连流量 a_i, 从 i 向 T 连流量 b_i, 求出最小割后就在每个元素的 AB 选项中做出了选择。

现在要加入额外代价, 即当 x_i 和 y_i 在不同集合时, 要让 S 到 T 仍然存在增广路, 可以发现 x_i 和 y_i 连双向流量 c_i 可以使其需要额外割掉这条边, 同时当 x_i 与 y_i 在同一个集合时这条边没有任何影响, 且不会导致某个点两个集合都选择。

[26] 网络流算法基本模型 - appgle https://www.cnblogs.com/hyl2000/p/6618519.html

[27] 关于平面图到对偶图的转化
https://blog.csdn.net/MaxMercer/article/details/77976666

[28] 网络流——最大权闭合子图
https://blog.csdn.net/Cold_Chair/article/details/52841351

变形: 若在同一个集合内还要付出代价 d_i(资源竞争), 目前想到的方法是假设已付出所有 d_i, 然后按照上文建图做 MCMF。

2.6 最小割树

2.6.1 Gusfield 算法构造

考虑单次求最小割的过程, 最小割将顶点集合一分为二, 设求 $u-v$ 的最小割 $cut(u,v)$, 顶点被分割为集合 U,V。

引理 2.16 $\forall x\in U, y\in V, cut(x,y)\le cut(u,v)$

证明 若存在 x,y 使得 $cut(x,y)>cut(u,v)$, 则 $cut(u,v)$ 无法把 x,y 分开, 与 $U\cap V=\emptyset$ 矛盾。

然后对每个点集再次选择两个点求最小割, 将其切分为两个集合, 直到所有集合都只有一个点为止。每次求完最小割后给 $u-v$ 连一条权重为 $cut(u,v)$ 的边, 这样做最后能得到一棵树。

定理 2.17 $u-v$ 在最小割树中的链上最小边权等于 $cut(u,v)$。

时间复杂度 $O(V^3E)$。

2.6.1.1 实现细节

Dinic 求出最小割后, 根据最后一次 BFS 找增广路时存储的层次标号是否有效对两个集合重新划分。若使用 ISAP 求最小割, 仍然需要使用 Dinic 的 BFS 划分集合。

2.6.2 询问

建出树后可以使用倍增法或线段树 + 树链剖分 $O(\lg n)$ 查询点对答案。

板子:

```
0 #include <algorithm>
  #include <cstdio>
  #include <cstring>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
      while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
10        c = getchar();
      }
      return res;
  }
  const int size = 505;
  namespace NetworkFlow {
      struct Edge {
          int to, nxt, f;
      } E[3005];
```

```
int last[size], cnt = 1;
void addEdge(int u, int v, int f) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u],
    E[cnt].f = f;
    last[u] = cnt;
}
int S, T, q[size], d[size], siz;
bool BFS() {
    memset(d + 1, -1, sizeof(int) * siz);
    int b = 0, e = 1;
    q[b] = S, d[S] = 0;
    while(b != e) {
        int u = q[b++];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(E[i].f && d[v] == -1) {
                d[v] = d[u] + 1;
                if(v == T)
                    return true;
                q[e++] = v;
            }
        }
    }
    return false;
}
int now[size];
int DFS(int u, int mf) {
    if(mf == 0 || u == T)
        return mf;
    int res = 0, k;
    for(int& i = now[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(d[v] == d[u] + 1 &&
           (k = DFS(v, std::min(mf, E[i].f)))) {
            E[i].f -= k, E[i ^ 1].f += k;
            mf -= k, res += k;
            if(mf == 0)
                break;
        }
    }
    if(res == 0)
        d[u] = -1;
    return res;
}
int dinic(int s, int t) {
    for(int i = 2; i <= cnt; i += 2) {
        int tot = E[i].f + E[i ^ 1].f;
        E[i].f = E[i ^ 1].f = tot >> 1;
    }
```

```
        S = s, T = t;
        int res = 0;
        while(BFS()) {
            memcpy(now + 1, last + 1,
                   sizeof(int) * siz);
            res += DFS(S, 1 << 30);
        }
        return res;
    }
    bool query(int u) {
        return d[u] == -1;
    }
}
namespace GHT {
    struct Edge {
        int to, nxt, w;
    } E[size * 2];
    int last[size], cnt = 0;
    void addEdge(int u, int v, int w) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u],
        E[cnt].w = w;
        last[u] = cnt;
    }
    int p[size][9], d[size], k[size][9];
    void DFS(int u) {
        for(int i = 1; i < 9; ++i) {
            int pp = p[u][i - 1];
            p[u][i] = p[pp][i - 1];
            k[u][i] =
                std::min(k[u][i - 1], k[pp][i - 1]);
        }
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(p[u][0] != v) {
                p[v][0] = u;
                k[v][0] = E[i].w;
                d[v] = d[u] + 1;
                DFS(v);
            }
        }
    }
    int query(int u, int v) {
        int res = 1 << 30;
        if(d[u] < d[v])
            std::swap(u, v);
        int delta = d[u] - d[v];
        for(int i = 0; i < 9; ++i)
            if(delta & (1 << i)) {
                res = std::min(res, k[u][i]);
```

```
                u = p[u][i];
            }
        if(u == v)
            return res;
        for(int i = 8; i >= 0; --i)
            if(p[u][i] != p[v][i]) {
                res = std::min(
                    res, std::min(k[u][i], k[v][i]));
                u = p[u][i], v = p[v][i];
            }
        return std::min(res,
                        std::min(k[u][0], k[v][0]));
    }
}
int A[size];
void buildGHTImpl(int l, int r) {
    if(l == r)
        return;
    int s = A[l], t = A[r];
    int w = NetworkFlow::dinic(s, t);
    int wp = l;
    for(int i = l; i <= r; ++i)
        if(NetworkFlow::query(A[i]))
            std::swap(A[wp++], A[i]);
    GHT::addEdge(s, t, w);
    GHT::addEdge(t, s, w);
    buildGHTImpl(l, wp - 1);
    buildGHTImpl(wp, r);
}
void buildGHT(int n) {
    NetworkFlow::siz = n;
    for(int i = 1; i <= n; ++i)
        A[i] = i;
    buildGHTImpl(1, n);
    srand(19260817);
    GHT::DFS(rand() % n + 1);
}
int main() {
    int n = read();
    int m = read();
    while(m--) {
        int u = read();
        int v = read();
        int w = read();
        NetworkFlow::addEdge(u, v, w);
        NetworkFlow::addEdge(v, u, w);
    }
    buildGHT(n);
    int q = read();
    while(q--)
```

```
        printf("%d\n", GHT::query(read(), read()));
    return 0;
}
```

上述内容参考了 UranusITS 的博客[29]。

2.7 技巧总结

2.7.1 最大流

- 若一个点只能被经过有限次, 将其拆为入点和出点, 入点到出点连流量为经过次数限制的边。
- 树形最大流可以贪心解决。
- 将大于等于限制转化为恰好等于限制, 由于最大流解决的限制是小于等于, 当其满流时恰好有解。
- 构图使得满流时的解就是合法解。
- 网格图若只需考虑相邻格子的影响, 考虑黑白染色, 然后建图 $S \to B \to W \to T$。

2.7.2 最小割

- 最大化收益可以理解为已经拿到所有收益, 最小化损失。然后将其转化为最小割解决。
- 使用 $+\infty$ 边描述依赖关系, 可以保证这条边不出现在最小割中。
- 用 S, T 与点的连边来表示点的权。

2.7.3 费用流

- 要求费用最小且边数最小: 类比进制的思想, 实际费用乘以一个大于总边数的因子, 再加上 1 作为该边边权。
- 若已知走一条边之前必定已经走完了另外几条边, 则考虑动态加边。
- 对于层数较少, 结构简单的图, 考虑使用其它数据结构贪心模拟费用流。**注意还要模拟退流(即反悔)。**
- (待验证)判断一条边是否一定被选: 在残量网络上跑 SPFA, 若距离差不等于边权则必选。
- 餐巾计划问题: 一条脏餐巾被传递多次只能代表 1 的流量, 会误导 MCMF。考虑修改建图方式: 一条餐巾被使用一次必须要直接从 S 流到 T, 脏餐巾只能从 S 重新流出, 由于每天会使用 r 条餐巾, 再从 S 免费补偿 r 条餐巾到表示当天将重复使用的脏餐巾的点(这一步尤为关键, 注意题目是否有这个性质)。简单地说, 就是如果要一个物品经过一条路径需要消耗大于 1 的流量时, 必须把每一段都拆成 S-T 的路径。

  ~~2019.3.5: 为什么我在 LOJ 上的餐巾计划和 Luogu 上的星际竞速代码都是 rank2 啊。。。~~

[29] [学习笔记] 最小割树(Gomory-Hu Tree) http://www.cnblogs.com/coder-Uranus/p/9771919.html

- 「雅礼集训 2018 Day8」B 乱搞：贪心策略是每次选取最长的路径优化，那么需要将时间当做费用，每次优化都选取路径上的费用最小值，将其当做流量限制。剩下部分的证明留坑待补。
- Codeforces739E Gosha is hunting：两种精灵球混合使用会多计算 p_iu_i 的期望，由于网络流并不支持流量缩放的功能，只能考虑最后减去这个值。那么精灵连两条边到汇点，一条费用为 0，另一条为 $-p_iu_i$。只使用一个精灵球时，MCMF 将贪心选择费用为 0 的边。简单地说，用流量和费用 + 贪心性质诱导 MCMF。
- SNOI2019 通信：$O(n^2)$ 连边过多，可以考虑分治连边，按照点值添加新点，然后让差值在新点连起来的链上累积，边数 $O(n \lg n)$。关键字：可累加的代价（距离绝对值）

上述内容参考了胡伯涛的 2007 年国家集训队论文《最小割模型在信息学竞赛中的应用》[3] 和租酥雨的博客[30]。

[30]网络流总结 https://www.cnblogs.com/zhoushuyu/p/8137534.html

Chapter 3

数据结构

3.1 树状数组 61
3.1.1 标号管辖范围 61
3.1.2 线性预处理 62
3.1.3 lowbit 函数原理 62
3.1.4 区间加法区间求和 62
3.2 线段树 62
3.2.1 技巧 62
3.2.2 zkw 线段树 63
3.2.3 势能分析线段树 63
3.2.4 Segment Tree Beats(吉司机线段树)与历史最值问题 . . . 63
3.2.5 线段树分治 69
3.2.6 线段树优化建图 73
3.2.7 猫树 76
3.2.8 线段树合并 83
3.2.9 线段树分裂 83
3.2.10 李超线段树 85
3.2.11 线段树维护单调栈 88
3.2.12 时间线段树 90
3.3 划分树 90
3.3.1 构建 91
3.3.2 查询 91
3.3.3 板子 91
3.4 二叉搜索树 93
3.4.1 二叉搜索树性质 93
3.4.2 FHQTreap 93
3.4.3 splay 97
3.5 动态树 99
3.5.1 常规操作 99
3.5.2 技巧/常见方法 102
3.6 并查集 111
3.6.1 路径压缩 111
3.6.2 按秩合并 111

3.6.3 复杂度证明 . . . 111
3.6.4 并查集的分裂 . . . 111
3.6.5 并查集重构树 . . . 113
3.7 K-D Tree . . . 115
3.7.1 构树 . . . 115
3.7.2 插入 . . . 116
3.7.3 删除 . . . 116
3.7.4 查询 . . . 116
3.7.5 估值 . . . 117
3.7.6 技巧 . . . 117
3.8 堆 . . . 118
3.8.1 左偏树 . . . 118
3.8.2 斜堆 . . . 119
3.8.3 可删堆 . . . 119
3.8.4 配对堆 . . . 120
3.9 可持久化数据结构 . . . 121
3.9.1 主席树 . . . 121
3.9.2 可持久化 Trie . . . 121
3.9.3 可持久化数组 . . . 121
3.9.4 优化 . . . 121
3.10 DLX 舞蹈链 . . . 124
3.10.1 X 算法 . . . 124
3.10.2 DLX . . . 125
3.11 Hash 表 . . . 128
3.11.1 链接法 . . . 128
3.11.2 双重散列 + 开放寻址法 . . . 128
3.11.3 完全散列 . . . 128
3.12 珂朵莉树/老司机树 . . . 128
3.12.1 存储 . . . 129
3.12.2 切分 . . . 129
3.12.3 区间赋值 . . . 129
3.12.4 应用 . . . 130
3.13 ETT . . . 130
3.13.1 欧拉遍历序列 . . . 130
3.13.2 基本操作 . . . 130

3.1 树状数组

树状数组用于支持区间加减法的数据维护。若只需要前缀和, 可不满足区间减法。

3.1.1 标号管辖范围

画图可发现, 节点 i 保存的是 $[i-lowbit(i)+1,i]$ 的和。在修改时, $+lowbit(i)$ 会使自己移动到父节点。在查询时 $-lowbit(i)$ 会使自己向前移动 $lowbit(i)$ 位, 离开自己的管辖范围, 移动到管辖范围前的更高层的节点。

事实上如果只需单点修改, 前缀查询, 即使信息不满足区间减法也可以使用树状数组。

3.1.2 线性预处理

精确地控制时序可以减小常数。

LinearBuild

```
for(int i=1;i<=n;++i) {
    int j=i+(i&-i);
    if(j<=n)A[j]+=A[i];
}
```

3.1.3 lowbit 函数原理

lowbit 函数定义为:$lowbit(i) = i\& - i$。

由于 i 始终为正,所以 $-i$ 的补码表示是 i 的位取反再加 1。因此末尾形如 0100 的位会变为 1111,可以发现 i 与 $-i$ 在末尾 1 前后的位均不同,该部分位与后为 0。因此 $i\& - i$ 仅保留末尾 1 的位。

3.1.4 区间加法区间求和

记原数组为 A_i,差分数组 $D_i = A_i - A_{i-1}$,则有 $A_i = \sum_{j=1}^{i} D_j$。将区间求和转化为前缀和之差,记前缀和为 S_i,有 $S_i = \sum_{j=1}^{i} A_j = \sum_{j=1}^{i} (i-j+1)D_j$。将该式拆为 $(i+1)\sum_{j=1}^{i} D_j - \sum_{j=1}^{i} jD_j$,做区间加法时在差分数组上打标记,同时维护数组 iD_i。

3.2 线段树

线段树用于支持快速区间加法的数据维护。

3.2.1 技巧

3.2.1.1 全局最优值剪枝

可以使用全局变量维护自己当前遍历到的最优值,若父节点维护的信息表明管辖范围内不可能出现更优值,则直接返回减少递归深度。(在 kd-tree 中比较有效)

3.2.1.2 局部最优值剪枝

例如维护区间 max 值时,顺便维护区间 min 值,modify 时发现操作无效直接退出。

3.2.1.3 不使用子树信息合并更新

比如单点加法,sum 值可以直接在下降时维护,而不用从子树信息 update。

3.2.1.4 标记永久化

直接将对整个区间的操作存到标记中而不下放, 统计时加回去, 减少空间占用(标记即值)与标记下放常数, 对可持久化友好。不过有些边界问题要小心处理。

3.2.1.5 代替平衡树

在一些简单的场合中, 可以使用值域线段树/树状数组完成平衡树的功能。对于不满足值域要求的, 使用离散化预处理。常数比平衡树小得多。

3.2.2 zkw 线段树

如果线段树维护的区间是 $[0, 2^N)$, 区间 $[b, e)$ 对应的子区间为 $[b, m), [m, e)$, 则该线段树是一棵 perfect binary tree。可以发现叶子节点的编号是对应原数组下标编号 $+2^N$。那么就可以非递归地找到叶子, 然后让叶子同步向上跳。

实际实现时先将闭区间变为开区间, 然后找到区间端点对应的叶子, 统计内部的区间, 再向上跳, 直到**两个节点是兄弟为止**。

3.2.2.1 简单操作

以区间求和为例:

```
0 off=1<<N;
  int res=0;
  for(s+=off-1,t+=off+1;s^t^1;s>>=1,t>>=1) {
      if(~s&1) res+=A[s^1];
      if(t&1) res+=A[t^1];
  }
```

单点修改直接从叶子节点开始向上跳。构造线段树时从 $off - 1$(最后一个非叶子节点)到 1 update。

由于需要转化为开区间, 实际支持的查询区间为 $[1, 2^N - 1)$, 最大空间仍旧为 4 倍原数组大小。

3.2.2.2 带标记操作

原先区间修改需要打标记, 不得不回到自顶向下的递归方法。使用标记永久化, 就可以支持自底向上修改与查询。有些情况下标记与值的意义是相同的。对于区间加法, 区间最值这类问题可以使用差分思想将值转化为自身的值与父亲之差使其可以标记永久化。

注意求区间最值时使用闭区间,原因暂时不明。

上述内容参考了 zkw 的课件《统计的力量——线段树全接触》。

3.2.3 势能分析线段树

对于某些无法打标记的区间操作(例如区间元素开根号), 若该操作对某个元素施加有限次操作就会使区间元素不再变化, 同样可以使用线段树。每次区间操作暴力修改, 合并时维护下次操作对区间内元素是否均无效。

限制还可以再宽松些, 如果该操作对区间内的元素施加的影响相同, 就可以打标记, 否则递归处理。~~考场上没时间分析复杂度, 一般是对的~~。

3.2.4 Segment Tree Beats(吉司机线段树)与历史最值问题

该线段树用于解决有类似 "将区间内小于 x 的数变为 x" 修改操作的问题。

3.2.4.1 主要思想

该线段树除了维护区间最小值外, 还维护了区间的严格次小值。那么当 $x \leq$ 最小值时, 直接退出; 当最小值 $< x <$ 次小值时 (注意如果条件为 $x \leq$ 次小值, 虽然取等号时仍然可以维护序列, 但是此时次小值变为了最小值, 无法维护新的次小值), 根据其它信息维护整个区间; 否则递归到子区间中处理。在序列规模为 n, 操作数为 m 时, 时间复杂度 $O(m \lg^2 n)$, 一般情况下该上界很松, 更接近 $O(m \lg n)$。若没有其它种类的区间修改操作 (比如区间加法), 时间复杂度为 $O(m \lg n)$。

3.2.4.2 细节

push 操作 这类操作不使用懒标记更新儿子。考虑区间取 max 操作, 如果在 push 时发现子区间的最小值比整个区间最小值更小, 就意味着整个区间执行过一次该操作, 此时对该子区间下放标记。

同时有取 max 与 min 操作 注意更新时区间内所有的值相同的情况, 比如更新 max 时可能会影响到 min 的值。push 时不必考虑两个操作的下传顺序。

维护多数组 每个位置在每个数组中有两个属性, 即是/不是最值。对每种属性组合的位置分别维护, 时空复杂度多了 $O(2^k)$ 因子, k 为数组数。

例题 BZOJ4695 最假女选手 按照套路维护区间最值, 次最值, 为了支持区间和的询问还要维护最值个数。update 时尽量减少判断以使代码简短, 易于 Debug。

参考代码:

```
#include <algorithm>
#include <cstdio>
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
const int size = 500005, inf = 1 << 30;
typedef long long Int64;
#define asInt64 static_cast<Int64>
struct Node {
    Int64 sum;
    int maxv, cmax, smaxv;
    int minv, cmin, sminv;
    int lazy;
} T[size << 2];
#define ls l, m, id << 1
```

```
#define rs m + 1, r, id << 1 | 1
void update(int id) {
    int l = id << 1, r = id << 1 | 1;
    T[id].sum = T[l].sum + T[r].sum;

    if(T[l].maxv == T[r].maxv) {
        T[id].maxv = T[l].maxv;
        T[id].cmax = T[l].cmax + T[r].cmax;
        T[id].smaxv = std::max(T[l].smaxv, T[r].smaxv);
    } else {
        if(T[l].maxv < T[r].maxv)
            std::swap(l, r);
        T[id].maxv = T[l].maxv;
        T[id].cmax = T[l].cmax;
        T[id].smaxv = std::max(T[l].smaxv, T[r].maxv);
    }

    if(T[l].minv == T[r].minv) {
        T[id].minv = T[l].minv;
        T[id].cmin = T[l].cmin + T[r].cmin;
        T[id].sminv = std::min(T[l].sminv, T[r].sminv);
    } else {
        if(T[l].minv > T[r].minv)
            std::swap(l, r);
        T[id].minv = T[l].minv;
        T[id].cmin = T[l].cmin;
        T[id].sminv = std::min(T[l].sminv, T[r].minv);
    }
}
void build(int l, int r, int id) {
    if(l == r) {
        T[id].sum = T[id].minv = T[id].maxv = read();
        T[id].cmin = T[id].cmax = 1;
        T[id].sminv = inf, T[id].smaxv = -inf;
    } else {
        int m = (l + r) >> 1;
        build(ls);
        build(rs);
        update(id);
    }
}
void color(int l, int r, int id, int val) {
    T[id].sum += asInt64(r - l + 1) * val;
    T[id].maxv += val, T[id].smaxv += val;
    T[id].minv += val, T[id].sminv += val;
    T[id].lazy += val;
}
void reset(int l, int r, int id, int val) {
    T[id].sum = asInt64(r - l + 1) * val;
    T[id].smaxv = -inf, T[id].sminv = inf;
```

```
    T[id].cmax = T[id].cmin = r - l + 1;
}
void chkMax(int l, int r, int id, int val) {
    T[id].sum +=
        asInt64(val - T[id].maxv) * T[id].cmax;
    T[id].maxv = val;
    T[id].minv = std::min(T[id].minv, val);
    if(T[id].maxv == T[id].minv)
        reset(l, r, id, val);
    else
        T[id].sminv = std::min(T[id].sminv, val);
}
void chkMin(int l, int r, int id, int val) {
    T[id].sum +=
        asInt64(val - T[id].minv) * T[id].cmin;
    T[id].minv = val;
    T[id].maxv = std::max(T[id].maxv, val);
    if(T[id].maxv == T[id].minv)
        reset(l, r, id, val);
    else
        T[id].smaxv = std::max(T[id].smaxv, val);
}
void push(int l, int r, int id) {
    int m = (l + r) >> 1;
    if(T[id].lazy) {
        color(ls, T[id].lazy);
        color(rs, T[id].lazy);
        T[id].lazy = 0;
    }
    if(T[id].maxv < T[id << 1].maxv)
        chkMax(ls, T[id].maxv);
    if(T[id].maxv < T[id << 1 | 1].maxv)
        chkMax(rs, T[id].maxv);
    if(T[id].minv > T[id << 1].minv)
        chkMin(ls, T[id].minv);
    if(T[id].minv > T[id << 1 | 1].minv)
        chkMin(rs, T[id].minv);
}
void add(int l, int r, int id, int nl, int nr,
         int val) {
    if(nl <= l && r <= nr)
        color(l, r, id, val);
    else {
        push(l, r, id);
        int m = (l + r) >> 1;
        if(nl <= m)
            add(ls, nl, nr, val);
        if(m < nr)
            add(rs, nl, nr, val);
        update(id);
```

```
    }
}
void CASMax(int l, int r, int id, int nl, int nr,
            int val) {
    if(T[id].maxv <= val)
        return;
    if(nl <= l && r <= nr && val > T[id].smaxv)
        chkMax(l, r, id, val);
    else {
        push(l, r, id);
        int m = (l + r) >> 1;
        if(nl <= m)
            CASMax(ls, nl, nr, val);
        if(m < nr)
            CASMax(rs, nl, nr, val);
        update(id);
    }
}
void CASMin(int l, int r, int id, int nl, int nr,
            int val) {
    if(T[id].minv >= val)
        return;
    if(nl <= l && r <= nr && val < T[id].sminv)
        chkMin(l, r, id, val);
    else {
        push(l, r, id);
        int m = (l + r) >> 1;
        if(nl <= m)
            CASMin(ls, nl, nr, val);
        if(m < nr)
            CASMin(rs, nl, nr, val);
        update(id);
    }
}
Int64 querySum(int l, int r, int id, int nl, int nr) {
    if(nl <= l && r <= nr)
        return T[id].sum;
    push(l, r, id);
    int m = (l + r) >> 1;
    Int64 res = 0;
    if(nl <= m)
        res += querySum(ls, nl, nr);
    if(m < nr)
        res += querySum(rs, nl, nr);
    return res;
}
int queryMin(int l, int r, int id, int nl, int nr) {
    if(nl <= l && r <= nr)
        return T[id].minv;
    push(l, r, id);
```

```
    int m = (l + r) >> 1;
    if(nl <= m && m < nr)
        return std::min(queryMin(ls, nl, nr),
                        queryMin(rs, nl, nr));
    if(nl <= m)
        return queryMin(ls, nl, nr);
    return queryMin(rs, nl, nr);
}
int queryMax(int l, int r, int id, int nl, int nr) {
    if(nl <= l && r <= nr)
        return T[id].maxv;
    push(l, r, id);
    int m = (l + r) >> 1;
    if(nl <= m && m < nr)
        return std::max(queryMax(ls, nl, nr),
                        queryMax(rs, nl, nr));
    if(nl <= m)
        return queryMax(ls, nl, nr);
    return queryMax(rs, nl, nr);
}
#define root 1, n, 1
int main() {
    int n = read();
    build(1, n, 1);
    int m = read();
    for(int i = 1; i <= m; ++i) {
        int op = read();
        int l = read();
        int r = read();
        switch(op) {
            case 1:
                add(root, l, r, read());
                break;
            case 2:
                CASMin(root, l, r, read());
                break;
            case 3:
                CASMax(root, l, r, read());
                break;
            case 4:
                printf("%lld\n", querySum(root, l, r));
                break;
            case 5:
                printf("%d\n", queryMax(root, l, r));
                break;
            case 6:
                printf("%d\n", queryMin(root, l, r));
                break;
        }
    }
```

```
    return 0;
}
```

3.2.4.3 区间最值问题转化为区间加减问题

注意到在上述方法区间取 min/max 打标记时, 只有区间最值受到影响。那么就可以将其转化为对节点管辖范围内最值的加减操作。在标记下传时仅传到受影响的儿子中。

3.2.4.4 历史最值问题

历史最值问题即在维护一个序列的同时, 支持询问指定位置从开始到当前时刻的最值/版本和, 甚至指定区间的历史最值的最值/和, 历史版本和之和。

区间加减 + 历史最值: 除了维护整体加减标记 add 外, 还维护整体加减标记从上一次标记下传到现在的最值 pre。那么节点 u 到儿子 v 的标记下传为 $pre_v = max(pre_v, add_v + pre_u), add_v += add_u$。

+ 区间覆盖:额外维护覆盖标记也可以完成标记下传与更新。

+ 区间最值/加减带 Clamp: 此时使用一种通用的标记 (a, b), 表示区间全体 $+a$, 再与 b 取最值。那么区间加减表示为 $(x, \pm\infty)$, 区间覆盖表示为 $(\pm\infty, x)$, 区间最值表示为 $(0, x)$, 区间加减带 Clamp 表示为 $(x, 0)$。以对 b 取 max 为例, 标记下传时 $b_v = max(b_v, b_u + a_v), a_v += a_u$。

+ 查询区间历史最值的最值:使用区间加减操作执行区间最值操作。

区间加减 + 区间历史最值之和:维护当前数组 A, 差值数组 $C = A - B$, 其中 B 为历史最值数组。那么区间加减可以转化为对数组 C 的区间加减带 Clamp 操作。两数组的区间和之和即为答案。

区间加减 + 区间历史版本和之和: 参见第 549页中 “ 支持有后效性的可加减区间操作, 询问历史版本和”。

+ 区间最值:同样按照是否为最值分类讨论。

上述内容参考了 santongding[1]与 Galaxies[2]的博客, 以及 C_SUNSHINE 和 jiry_2 的营员交流课件《Segment ree Beats!》, 还有吉如一的 2016 年国家集训队论文《区间最值操作与历史最值问题》。

3.2.5 线段树分治

对于可离线线段树套其它数据结构的树套树问题, 有时会因为空间不够导致无法使用树套树, 并且树套树代码量大, 不易调试。

既然这类问题可离线, 可以考虑将所有询问同时在线段树上移动。对询问进行分治, 每次处理分治区间时, 先建出查询数据结构, 然后对完全包含这段区间的询问进行查询并更新答案, 再清空数据结构, 将其它询问分治到左右子区间中求解。使用分治可以保证任意时刻只有单个区间的查询数据结构。

3.2.5.1 例题 [FJOI2015] 火星商店问题

典型的线段树套可持久化 Trie 问题, 使用线段树分治毫无空间压力, 时间复杂度与原做法相同。

参考代码:

[1]吉司机线段树 (segment tree beats!)
https://blog.csdn.net/qq_36284842/article/details/80058746

[2]bzoj4695 最假女选手 https://www.cnblogs.com/galaxies/p/bzoj4695.html

```
#include <algorithm>
#include <cstdio>
namespace IO {
    const int bsiz = 1 << 23;
    char in[bsiz], *ip = in;
    void init() {
        fread(in, bsiz, 1, stdin);
    }
    char getchar() {
        return *(ip++);
    }
    char out[bsiz], *op = out;
    void putchar(char ch) {
        *(op++) = ch;
    }
    void flush() {
        fwrite(out, op - out, 1, stdout);
    }
}
int read() {
    int res = 0, c;
    do
        c = IO::getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = IO::getchar();
    }
    return res;
}
void write(int x) {
    if(x >= 10)
        write(x / 10);
    IO::putchar('0' + x % 10);
}
const int size = 100005, maxd = 16;
struct Node {
    int ch[2], siz;
} T[size * 20];
int icnt = 0, root[size];
void insert(int* src, int val) {
    for(int d = maxd; d >= -1; --d) {
        int id = ++icnt;
        T[id] = T[*src];
        ++T[id].siz;
        *src = id;
        if(d >= 0)
            src = &T[id].ch[(val >> d) & 1];
    }
}
```

```
int query(int b, int e, int val) {
    int res = 0;
    for(int d = maxd; d >= 0; --d) {
        int c = (val >> d) & 1, nc = c ^ 1;
        int dsiz =
            T[T[e].ch[nc]].siz - T[T[b].ch[nc]].siz;
        if(dsiz) {
            b = T[b].ch[nc], e = T[e].ch[nc];
            res |= 1 << d;
        } else
            b = T[b].ch[c], e = T[e].ch[c];
    }
    return res;
}
struct Operator {
    int op, l, r, id, v, b, e;
    // op=0:id,v,e
    // op=1:l,r,id,v,b,e
} Op[size];
int ans[size], q[size], sid[size];
bool cmp(int a, int b) {
    return Op[a].id < Op[b].id;
}
void solve(int l, int r, int b, int e) {
    if(l > r)
        return;
    icnt = 0;
    int qcnt = 0;
    for(int i = l; i <= r; ++i)
        if(Op[i].op == 0 && b <= Op[i].e &&
           Op[i].e <= e)
            q[qcnt++] = i;
    if(qcnt == 0)
        return;
    std::sort(q, q + qcnt, cmp);
    int scnt = 0;
    for(int i = 0; i < qcnt; ++i) {
        int cop = q[i];
        if(sid[scnt] != Op[cop].id) {
            sid[++scnt] = Op[cop].id;
            root[scnt] = root[scnt - 1];
        }
        insert(&root[scnt], Op[cop].v);
    }
    for(int i = l; i <= r; ++i)
        if(Op[i].op == 1 && Op[i].b <= b &&
           e <= Op[i].e) {
            int lp = std::lower_bound(sid + 1,
                                      sid + scnt + 1,
                                      Op[i].l) -
```

```
                sid - 1;
            int rp = std::upper_bound(sid + 1,
                                      sid + scnt + 1,
                                      Op[i].r) -
                sid - 1;
            int& res = ans[Op[i].id];
            res =
                std::max(res, query(root[lp], root[rp],
                                    Op[i].v));
        }
    if(b != e) {
        int m = (b + e) >> 1, cpos = l;
        for(int i = l; i <= r; ++i)
            if(Op[i].b <= m &&
               !(Op[i].b <= b && e <= Op[i].e))
                std::swap(Op[i], Op[cpos++]);
        solve(l, cpos - 1, b, m);
        cpos = l;
        for(int i = l; i <= r; ++i)
            if(m < Op[i].e &&
               !(Op[i].b <= b && e <= Op[i].e))
                std::swap(Op[i], Op[cpos++]);
        solve(l, cpos - 1, m + 1, e);
    }
}
int main() {
    IO::init();
    int n = read();
    int m = read();
    for(int i = 1; i <= n; ++i) {
        root[i] = root[i - 1];
        insert(&root[i], read());
    }
    int d = 1, qid = 0;
    for(int i = 1; i <= m; ++i) {
        int op = read();
        if(op) {
            Op[i].op = 1;
            Op[i].l = read();
            Op[i].r = read();
            Op[i].id = ++qid;
            Op[i].v = read();
            ans[qid] = query(root[Op[i].l - 1],
                             root[Op[i].r], Op[i].v);
            Op[i].b = d - read() + 1;
        } else {
            if(i != 1)
                ++d;
            Op[i].op = 0;
            Op[i].id = read();
```

```
            Op[i].v = read();
            Op[i].b = d;
        }
        Op[i].e = d;
    }
    solve(1, m, 1, d);
    for(int i = 1; i <= qid; ++i) {
        write(ans[i]);
        IO::putchar('\n');
    }
    IO::flush();
    return 0;
}
```

3.2.6 线段树优化建图

对于一个点到一个或多个连续区间内的点有连边且区间内的边权相等，考虑使用线段树优化建图。即使用线段树的上层节点代表管辖区间内的所有节点，建树时上层节点向左右儿子连权值为 0 的边（**注意上层节点区分入点与出点，与儿子连有向边**），建图时类似 modify 操作连边，边数由 $O(mn)$ 降为 $O(m \lg n)$。上层节点起到了“捆线带”的作用。

例题：CF786B Legacy

线段树优化建图 + 裸最短路。

```
#include <cstdio>
#include <cstring>
#include <queue>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005, psiz = size << 3;
struct Edge {
    int to, nxt, w;
} E[size * 40];
int last[psiz], cnt = 0;
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].w = w;
    last[u] = cnt;
}
#define ls l, m, id << 1
#define rs m + 1, r, id << 1 | 1
int iid[size << 2], oid[size << 2], pid[size],
```

```
    icnt = 0;
void build(int l, int r, int id) {
    if(l == r) {
        pid[l] = iid[id] = oid[id] = ++icnt;
    } else {
        iid[id] = ++icnt;
        oid[id] = ++icnt;
        int m = (l + r) >> 1;
        build(ls);
        build(rs);
        addEdge(iid[id], iid[id << 1], 0);
        addEdge(iid[id], iid[id << 1 | 1], 0);
        addEdge(oid[id << 1], oid[id], 0);
        addEdge(oid[id << 1 | 1], oid[id], 0);
    }
}
typedef void (*Func)(int u);
int nl, nr, p, w;
void funcIn(int u) {
    addEdge(p, iid[u], w);
}
void funcOut(int u) {
    addEdge(oid[u], p, w);
}
template <Func func>
void insert(int l, int r, int id) {
    if(nl <= l && r <= nr)
        func(id);
    else {
        int m = (l + r) >> 1;
        if(nl <= m)
            insert<func>(ls);
        if(m < nr)
            insert<func>(rs);
    }
}
typedef long long Int64;
Int64 dis[psiz];
int q[psiz];
struct Info {
    int u;
    Int64 dis;
    Info(int u, Int64 dis) : u(u), dis(dis) {}
    bool operator<(const Info& rhs) const {
        return dis > rhs.dis;
    }
};
void SSSP(int s) {
    memset(dis + 1, 0x3f, sizeof(Int64) * icnt);
    dis[s] = 0;
```

```
    std::priority_queue<Info> heap;
    heap.push(Info(s, 0));
    while(heap.size()) {
        int u = heap.top().u;
        Int64 cd = heap.top().dis;
        heap.pop();
        if(dis[u] != cd)
            continue;
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            Int64 nd = cd + E[i].w;
            if(dis[v] > nd) {
                dis[v] = nd;
                heap.push(Info(v, nd));
            }
        }
    }
}
int main() {
    int n = read();
    int q = read();
    int s = read();
    build(1, n, 1);
    while(q--) {
        int t = read();
        if(t == 1) {
            int u = read();
            int v = read();
            int w = read();
            addEdge(pid[u], pid[v], w);
        } else {
            p = pid[read()];
            nl = read();
            nr = read();
            w = read();
            if(t == 2)
                insert<funcIn>(1, n, 1);
            else
                insert<funcOut>(1, n, 1);
        }
    }
    SSSP(pid[s]);
    for(int i = 1; i <= n; ++i) {
        int u = pid[i];
        Int64 d = dis[u];
        if(d == 0x3f3f3f3f3f3f3f3f)
            d = -1;
        printf("%lld ", d);
    }
    return 0;
```

```
}
```

3.2.7 猫树

当无修改,区间查询次数多,区间信息仅支持结合律与快速合并时(不可减,不可重叠),猫树可以以更多预处理时间与空间的代价使单次查询复杂度变为 $O(1)$。

这种情况下一般使用线段树解决,但是线段树查询时需要合并 $O(\lg n)$ 次信息,考虑如何将合并次数降到 $O(1)$。以下不讨论 $l = r$ 的平凡情况。由于区间端点 l, r 分别在线段树上对应了某个节点的 mid,可以考虑预处理所有节点的 mid 到管辖范围中每个点的信息(左右方向在 m 处一开一闭),查询时快速找到某个 mid 满足 l, r 在它的管辖区间内且分别在它的两侧,利用预处理的 $[l, mid)$ 与 $[mid, r]$ 的信息,$O(1)$ 合并回答询问。

接下来考虑如何寻找 mid 对应的线段树节点 id。事实上这个节点就是 l, r 对应节点的 LCA。**猫树使用的是 zkw 线段树的节点标号方式,**l, r 的节点编号的 LCP 就是 mid 的节点编号。使用位运算 $id_l >> (1 + \lg(id_l \oplus id_r))$ 可以实现 $O(1)$ 求 LCP。

参考代码(LOJ#6057.「HNOI2016」序列 数据加强版):

```
#include <algorithm>
#include <cstdio>
typedef long long Int64;
#define asInt64 static_cast<Int64>
namespace IO {
    int A, B, C, P;
    Int64 lastAns = 0;
    inline int rnd() {
        return A = (A * B +
                    (C ^
                     (int)(lastAns & 0x7fffffffLL)) %
                        P) %
               P;
    }
    void init() {
        scanf("%d%d%d%d", &A, &B, &C, &P);
    }
}
const int size = 1 << 17, mod = 1000000007;
struct Data {
    Int64 sumAA, sumAB, pms;
    int rank;
} P[size * 20], *ap = P;
Data* alloc(int l, int r) {
    Data* res = ap - l;
    ap = res + r;
    return res;
}
struct Node {
    int m;
    Data* data;
    Int64 query(int l, int r) const {
        Int64 res = data[l].sumAA + data[r].sumAA;
        int lrk = data[l].rank, rrk = data[r].rank;
```

```
        if(lrk < rrk) {
            res += data[l].sumAB;
            Int64 lcnt = m - l, rcnt = lrk - lcnt;
            Int64 rsum = data[r].pms -
                (rcnt ? data[m + rcnt - 1].pms : 0);
            res += lcnt * rsum;
        } else {
            res += data[r].sumAB;
            Int64 rcnt = r - m + 1, lcnt = rrk - rcnt;
            Int64 lsum = data[l].pms -
                (lcnt ? data[m - lcnt].pms : 0);
            res += rcnt * lsum;
        }
        return res;
    }
} T[size << 1];
int A[size], B[size], n;
std::pair<int, int> st[size];
Int64 psum[size];
//[l,r)
void build(int l, int r, int id) {
    if(l + 1 == r || n <= l)
        return;
    int m = (l + r) >> 1;
    build(l, m, id << 1);
    build(m, r, id << 1 | 1);
    T[id].m = m;
    T[id].data = alloc(l, r);
    int top = 0, minv = 1 << 30;
    Int64 cans = 0, pms = 0;
    st[top].second = m;
    for(int i = m - 1; i >= l; --i) {
        int val = A[i];
        minv = std::min(minv, val);
        B[i] = minv;
        pms += minv;
        T[id].data[i].pms = pms;
        while(top && st[top].first >= val)
            --top;
        Int64 mul = asInt64(st[top].second - i) * val;
        ++top;
        st[top] = std::make_pair(val, i);
        psum[top] = psum[top - 1] + mul;
        cans += psum[top];
        T[id].data[i].sumAA = cans;
    }
    top = cans = pms = 0, minv = 1 << 30,
    st[top].second = m - 1;
    for(int i = m; i < r; ++i) {
        int val = A[i];
```

```
            minv = std::min(minv, val);
            B[i] = minv;
            pms += minv;
            T[id].data[i].pms = pms;
            while(top && st[top].first >= val)
                --top;
            Int64 mul = asInt64(i - st[top].second) * val;
            ++top;
            st[top] = std::make_pair(val, i);
            psum[top] = psum[top - 1] + mul;
            cans += psum[top];
            T[id].data[i].sumAA = cans;
        }
        int lp = m - 1, rp = m, crk = 0;
        cans = 0;
        while(l <= lp || rp < r) {
            if(rp >= r || (l <= lp && B[lp] > B[rp])) {
                T[id].data[lp].rank = ++crk;
                Int64 rcnt = crk - (m - lp);
                cans += rcnt * B[lp];
                T[id].data[lp].sumAB = cans;
                --lp;
            } else {
                T[id].data[rp].rank = ++crk;
                Int64 lcnt = crk - (rp - m + 1);
                cans += lcnt * B[rp];
                T[id].data[rp].sumAB = cans;
                ++rp;
            }
        }
    }
    const int maxb = 10, maxp = 1 << maxb;
    int lg2[maxp];
    int getLog(int x) {
        return x < maxp ? lg2[x] : maxb + lg2[x >> maxb];
    }
    Int64 query(int l, int r, int N) {
        if(l == r)
            return A[l];
        else {
            int pl = l + N, pr = r + N;
            int lca = pl >> (getLog(pl ^ pr) + 1);
            return T[lca].query(l, r);
        }
    }
    int main() {
        for(int i = 2; i < maxp; ++i)
            lg2[i] = lg2[i >> 1] + 1;
        int q;
        scanf("%d%d", &n, &q);
```

```
    for(int i = 0; i < n; ++i)
        scanf("%d", &A[i]);
    IO::init();
    int N = 1;
    while(N < n)
        N <<= 1;
    build(0, N, 1);
    int ans = 0;
    for(int i = 0; i < q; ++i) {
        int l = IO::rnd() % n, r = IO::rnd() % n;
        if(l > r)
            std::swap(l, r);
        IO::lastAns = query(l, r, N);
        ans = (ans + IO::lastAns) % mod;
    }
    printf("%d\n", (ans + mod) % mod);
    return 0;
}
```

3.2.7.1 扩展

虽然必须 “无修改”, 但是猫树结合二进制分组可以支持 “向后插入”, 可以用来优化 DP。

对于树上链信息询问, 如果使用 DFS 序 + 树链剖分会带来 $O(\lg n)$ 的复杂度。考虑点分治, 预处理重心到子树内每个点的链信息。由于区间信息合并不可重叠, 要分别预处理包括重心与不包括重心的信息。查询链对应的重心时在点分树上查询 LCA。为了得到 $O(1)$ 的查询复杂度, 需要 $O(1)$LCA 与 $O(1)$hashTable 求节点到重心的信息对应位置。

参考代码(LOJ#2013.「SCOI2016」幸运数字):

这题使用点分治 + 猫树有点杀鸡用牛刀的感觉。因为本题可离线。

```
#include <algorithm>
#include <cstdio>
#include <unordered_map>
typedef long long Int64;
Int64 read() {
    Int64 res = 0;
    int c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 20005;
struct G {
    struct Edge {
        int to, nxt;
```

```
    } E[size * 2];
    int last[size], cnt;
    G() : cnt(0) {}
    void addEdge(int u, int v) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u];
        last[u] = cnt;
    }
} g1, g2;
struct Base {
    Int64 b[64];
    void insert(Int64 x) {
        for(int i = 60; i >= 0; --i)
            if(x & (1LL << i)) {
                if(b[i])
                    x ^= b[i];
                else {
                    b[i] = x;
                    break;
                }
            }
    }
    Int64 maxv() const {
        Int64 res = 0;
        for(int i = 60; i >= 0; --i)
            res = std::max(res, res ^ b[i]);
        return res;
    }
};
Base merge(Base a, const Base& b) {
    for(int i = 60; i >= 0; --i)
        if(b.b[i])
            a.insert(b.b[i]);
    return a;
}
Base B[size * 16];
int bcnt = 0;
std::unordered_map<int, int> fap[size];
Int64 g[size];
bool vis[size];
int nrt, msiz, tsiz, siz[size];
void getRootImpl(int u, int p) {
    siz[u] = 1;
    int ssiz = 0;
    for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
        int v = g1.E[i].to;
        if(!vis[v] && v != p) {
            getRootImpl(v, u);
            siz[u] += siz[v];
            ssiz = std::max(ssiz, siz[v]);
```

```
        }
    }
    ssiz = std::max(ssiz, tsiz - siz[u]);
    if(ssiz < msiz)
        msiz = ssiz, nrt = u;
}
int getRoot(int u, int siz) {
    msiz = 1 << 30, tsiz = siz;
    getRootImpl(u, 0);
    return nrt;
}
void DFS(int u, int rt, int p, const Base& base) {
    int id = ++bcnt;
    fap[u][rt] = id;
    B[id] = base;
    B[id].insert(g[u]);
    for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
        int v = g1.E[i].to;
        if(!vis[v] && v != p)
            DFS(v, rt, u, B[id]);
    }
}
void divide(int u) {
    vis[u] = true;
    DFS(u, u, 0, Base());
    for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
        int v = g1.E[i].to;
        if(!vis[v]) {
            int rt = getRoot(v, siz[v]);
            g2.addEdge(u, rt);
            divide(rt);
        }
    }
}
int d[size], L[size], A[16][size * 2], icnt = 0;
void scan(int u) {
    L[u] = ++icnt;
    A[0][icnt] = u;
    for(int i = g2.last[u]; i; i = g2.E[i].nxt) {
        int v = g2.E[i].to;
        d[v] = d[u] + 1;
        scan(v);
        A[0][++icnt] = u;
    }
}
int lg2[1024];
int choose(int a, int b) {
    return d[a] < d[b] ? a : b;
}
void initLCA(int rt) {
```

```
    for(int i = 2; i < 1024; ++i)
        lg2[i] = lg2[i >> 1] + 1;
    scan(rt);
    for(int i = 1; i < 16; ++i) {
        int end = icnt - (1 << i) + 1, off = 1
            << (i - 1);
        for(int j = 1; j <= end; ++j)
            A[i][j] =
                choose(A[i - 1][j], A[i - 1][j + off]);
    }
}
int getLog(int x) {
    return x < 1024 ? lg2[x] : 10 + lg2[x >> 10];
}
int getLCA(int u, int v) {
    int l = L[u], r = L[v];
    if(l > r)
        std::swap(l, r);
    int p = getLog(r - l + 1);
    return choose(A[p][l], A[p][r - (1 << p) + 1]);
}
int main() {
    int n = read();
    int q = read();
    for(int i = 1; i <= n; ++i)
        g[i] = read();
    for(int i = 1; i < n; ++i) {
        int u = read();
        int v = read();
        g1.addEdge(u, v);
        g1.addEdge(v, u);
    }
    int rt = getRoot(1, n);
    divide(rt);
    initLCA(rt);
    for(int i = 0; i < q; ++i) {
        int u = read();
        int v = read();
        int lca = getLCA(u, v);
        Base &ud = B[fap[u][lca]],
             &vd = B[fap[v][lca]];
        Base ch = merge(ud, vd);
        printf("%lld\n", ch.maxv());
    }
    return 0;
}
```

上述内容参考了 immortalCO 的博客[3]。

[3]一种高效处理无修改区间或树上询问的数据结构(附代码)http://immortalco.blog.uoj.ac/blog/2102

3.2.8 线段树合并

该方法适用于总插入数(叶子数)为定值,且父线段树可以由子线段树的信息合并的情况。合并两棵子树时,若两棵子树都非空,则信息相加并返回某个节点(若需要可持久化则新开节点),否则返回有效的节点编号。

记值域为 n,总共有 m 棵含单个元素的线段树合并。该方法的时间复杂度为 $O(m \lg n)$,理由是该方法的复杂度不会比将元素逐个插入一棵空线段树更差。

该方法参考了黄嘉泰的课件《线段树的合并——不为人知的实用技巧》。

3.2.9 线段树分裂

需要将权值线段树的前 k 小与其余部分分为两棵线段树。一般用于区间排序。

类似于 FHQTreap,分裂时按照左子树的大小决定分裂的位置。实现中在原树上修改,分离出新的树。分离出的部分存储在新分配的节点中,单次分裂新增 $O(\lg n)$ 个节点。

线段树分裂的复杂度类似于 $O((n+m)\lg n)$ 的形式,即使加入线段树合并也是如此。

例题:[HEOI2016/TJOI2016] 排序

使用 set 维护有序区间(注意要同时维护升/降序),需要跨区间排序时就分裂出左右端对应有序区间的一部分,与中间有序区间的线段树合并成为新的有序区间。这个方法支持在线多次询问。

参考代码:

```
#include <cstdio>
#include <set>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005;
typedef std::set<int>::iterator IterT;
struct Node {
    int l, r, siz;
} T[size * 50];
int tcnt = 0;
int insert(int l, int r, int p) {
    int id = ++tcnt;
    T[id].siz = 1;
    if(l != r) {
        int m = (l + r) >> 1;
        if(p <= m)
            T[id].l = insert(l, m, p);
        else
            T[id].r = insert(m + 1, r, p);
    }
    return id;
```

```
}
int query(int l, int r, int id) {
    if(l == r)
        return l;
    int m = (l + r) >> 1;
    return T[T[id].l].siz ? query(l, m, T[id].l) :
                            query(m + 1, r, T[id].r);
}
int merge(int a, int b) {
    if(a && b) {
        T[a].siz += T[b].siz;
        T[a].l = merge(T[a].l, T[b].l);
        T[a].r = merge(T[a].r, T[b].r);
        return a;
    }
    return a | b;
}
int split(int src, int k, int d) {
    if(T[src].siz == k)
        return 0;
    int id = ++tcnt;
    T[id].siz = T[src].siz - k;
    T[src].siz = k;
    if(d) {
        int rsiz = T[T[src].r].siz;
        if(rsiz >= k) {
            T[id].r = split(T[src].r, k, d);
            T[id].l = T[src].l;
            T[src].l = 0;
        } else
            T[id].l = split(T[src].l, k - rsiz, d);
    } else {
        int lsiz = T[T[src].l].siz;
        if(lsiz >= k) {
            T[id].l = split(T[src].l, k, d);
            T[id].r = T[src].r;
            T[src].r = 0;
        } else
            T[id].r = split(T[src].r, k - lsiz, d);
    }
    return id;
}
int rt[size], dir[size];
std::set<int> interval;
//[pos,)
IterT split(int pos) {
    IterT it = interval.lower_bound(pos);
    if(*it == pos)
        return it;
    --it;
```

```
    rt[pos] = split(rt[*it], pos - *it, dir[*it]);
    dir[pos] = dir[*it];
    return interval.insert(pos).first;
}
int main() {
    int n = read();
    int m = read();
    for(int i = 1; i <= n; ++i) {
        rt[i] = insert(1, n, read());
        interval.insert(i);
    }
    interval.insert(n + 1);
    for(int i = 1; i <= m; ++i) {
        int d = read();
        int l = read();
        int r = read();
        IterT beg = split(l);
        IterT end = split(r + 1);
        int crt = 0;
        for(IterT it = beg; it != end; ++it)
            crt = merge(crt, rt[*it]);
        rt[l] = crt;
        dir[l] = d;
        interval.erase(++beg, end);
    }
    int q = read();
    split(q);
    split(q + 1);
    printf("%d\n", query(1, n, rt[q]));
    return 0;
}
```

该内容参考了 FlashHu 的博客[4]。

仅合并与分裂时尽量使用线段树合并分裂而不是 FHQTreap，后者常数太大。

3.2.10 李超线段树

李超线段树用来维护多个线段在指定整点上的最值，一般用于斜率优化维护动态凸包。该方法常数小，细节少。其关键在于标记永久化，只要保证最优解在链上就可以了。每个区间对应的节点存储在覆盖整个区间的线段中，在区间中点处最高/低的线段。查询时计算链上存储的线段在查询点上的值的最值。

实现时在 *color* 函数（此时线段完全覆盖区间，可以当做直线处理）中处理新加入的线段：

- 计算出两条线段在左右端点上的函数值。
- 若当前线段完全覆盖原线段，则用当前线段替代并返回。
- 反之若当前线段被原线段完全覆盖，则直接返回。

[4] 有趣的线段树模板合集（线段树，最短/长路，单调栈，线段树合并，线段树分裂，树上差分，Tarjan-LCA，势能线段树，李超线段树）
https://www.cnblogs.com/flashhu/p/9651161.html

- 否则两条线段不平行且交点在区间内, 计算出两条直线的交点 p。
- 则向交点所在的半边推送在该半边可能取最值即将被抛弃的线段。
- 尝试更新在当前区间中点上取得最值的线段。

由于单次 *color* 可能向一条链推送线段, 时间复杂度为 $O(n \lg^2 n)$。

正确性证明 主要思路是证明在执行一次修改操作后正确性不变。

前两种提前返回的情况显然正确。在第三种情况中, 假设有两条相交线段 A, B, 中点 m 在 A 处得到更优值, 对于交点所在部分另一边的任意一点, 它们都在 A 处取得更优值。因此将线段 A 当做当前区间的节点的永久化标记。对于交点所在部分的区间, 它与 A 孰优孰劣并不清楚, 索性先保留 A, 将被抛弃的 B 下放到儿子处理。如果该边的点在 A 处取得最值, 因为 A 在链上, 必定会被计算。如果在 B 处取得最值, 由于 B 的信息被保留到后代, 也能被计算。

参考代码(「雅礼集训 2017 Day2」线段游戏):

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
const int size = 100000;
typedef double FT;
const FT inf = 1e20;
#define asFT static_cast<FT>
struct Line {
    FT k, b;
    FT operator()(FT x) const {
        return k * x + b;
    }
} L[(size + 5) << 2];
#define ls l, m, id << 1
#define rs m + 1, r, id << 1 | 1
void build(int l, int r, int id) {
    L[id].b = -inf;
    if(l != r) {
        int m = (l + r) >> 1;
        build(ls);
        build(rs);
```

```
    }
}
void color(int l, int r, int id, const Line& B) {
    Line& A = L[id];
    FT la = A(l), ra = A(r), lb = B(l), rb = B(r);
    if(la >= lb && ra >= rb)
        return;
    if(la <= lb && ra <= rb) {
        A = B;
        return;
    }
    int m = (l + r) >> 1;
    FT pos = (A.b - B.b) / (B.k - A.k);
    if(pos <= m)
        color(ls, la > lb ? A : B);
    else
        color(rs, ra > rb ? A : B);
    if(pos <= m ? rb > ra : lb > la)
        A = B;
}
void insert(int l, int r, int id, int nl, int nr,
            const Line& line) {
    if(nl <= l && r <= nr)
        color(l, r, id, line);
    else {
        int m = (l + r) >> 1;
        if(nl <= m)
            insert(ls, nl, nr, line);
        if(m < nr)
            insert(rs, nl, nr, line);
    }
}
FT query(int l, int r, int id, int p) {
    FT res = L[id](p);
    if(l != r) {
        int m = (l + r) >> 1;
        if(p <= m)
            res = std::max(res, query(ls, p));
        else
            res = std::max(res, query(rs, p));
    }
    return res;
}
void addSeg() {
    int x1 = read();
    int y1 = read();
    int x2 = read();
    int y2 = read();
    if(std::max(x1, x2) < 1 || std::min(x1, x2) > size)
        return;
```

```
        Line line;
        if(x1 == x2) {
            line.k = 0.0;
            line.b = std::max(y1, y2);
        } else {
            line.k = (y1 - y2) / asFT(x1 - x2);
            line.b = y1 - line.k * x1;
        }
        if(x1 > x2)
            std::swap(x1, x2);
        insert(1, size, 1, x1, x2, line);
    }
    int main() {
        int n = read();
        int m = read();
        build(1, size, 1);
        for(int i = 1; i <= n; ++i)
            addSeg();
        for(int i = 1; i <= m; ++i) {
            if(read()) {
                int x0 = read();
                FT res = query(1, size, 1, x0);
                if(res < -1e19)
                    puts("0");
                else
                    printf("%.3lf\n", res);
            } else
                addSeg();
        }
        return 0;
    }
```

底层优化：C++11 中支持 std::fma，可以单次完成乘加运算，速度更快且精度更高。不过当 #pragma STDC FP_CONTRACT 开关打开时，编译器会自行优化。

剪枝优化：插入线段而不是插入直线时，将 B 无贡献的剪枝提前到 modify 而不是 color。

「SDOI2016」游戏：虽然树链剖分后不能保证区间内的横坐标单调，但可以保证插入和查询的区间横坐标单调，color 函数的正确性并不会受到影响。因此不必考虑这个问题。

~~2019.3.8：为什么又是 rank2。。。~~

该内容参考了 Mangoyang 的博客[5]。

3.2.11 线段树维护单调栈

动态维护单调栈，每次修改后回答单调栈的信息（栈顶/栈大小）。

使用线段树维护单调栈，关键在于两个子区间单调栈的合并。假设要维护一个单调上升的栈，在合并左右区间时，右区间的左部可能会被丢弃，递归二分右区间找到截断点再维护其它信息。在二分时加入剪枝还可以提高速度：

- 若二分值比区间左端点小，则返回整个区间的信息。

[5]「学习笔记」李超线段树 http://www.cnblogs.com/mangoyang/p/9979465.html

- 若二分值比区间最值大, 则返回空。

参考代码(LuoguP4198 楼房重建):

```
#include <algorithm>
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005;
typedef long long Int64;
#define asInt64 static_cast<Int64>
int siz[size << 2], h[size];
struct Slope {
    int X, Y;
    Slope() : X(1), Y(0) {}
    Slope(int X, int Y) : X(X), Y(Y) {}
    bool operator<(const Slope& rhs) const {
        return asInt64(X) * rhs.Y > asInt64(rhs.X) * Y;
    }
    bool operator<=(const Slope& rhs) const {
        return asInt64(X) * rhs.Y >=
            asInt64(rhs.X) * Y;
    }
} slope[size << 2];
#define ls l, m, id << 1
#define rs m + 1, r, id << 1 | 1
int split(int l, int r, int id, Slope k) {
    if(slope[id] <= k)
        return 0;
    if(l == r)
        return 1;
    if(k < Slope(l, h[l]))
        return siz[id];
    int m = (l + r) >> 1;
    if(slope[id << 1] < k)
        return split(rs, k);
    return split(ls, k) + siz[id] - siz[id << 1];
}
void insert(int l, int r, int id, Slope k) {
    if(l == r)
        slope[id] = k, siz[id] = 1;
    else {
```

```
        int m = (l + r) >> 1;
        if(k.X <= m)
            insert(ls, k);
        else
            insert(rs, k);
        slope[id] = std::max(slope[id << 1],
                             slope[id << 1 | 1]);
        siz[id] =
            siz[id << 1] + split(rs, slope[id << 1]);
    }
}
int main() {
    int n = read();
    int m = read();
    for(int i = 1; i <= m; ++i) {
        int x = read();
        int y = read();
        h[x] = y;
        insert(1, n, 1, Slope(x, y));
        printf("%d\n", siz[1]);
    }
    return 0;
}
```

3.2.12 时间线段树

例题 BZOJ 4184 shallot

该数据结构仅支持插入, 每次向该数据结构插入或删除一个元素, 求每个时刻的数据结构。

数据结构的限制使得我们必须只使用插入操作, 考虑将元素的插入删除操作看做是元素存在于一段区间(有趣的是, 有时候我们将元素存在于一段区间拆分为插入和删除, 即扫描线思想)。将时间轴上的数据结构的继承关系解除, 即假设它们刚开始都是空的, 那么就可以将其转化为向时间 $[b, e]$ 内的所有数据结构插入该元素。

此时区间操作可以使用线段树分治的思想优化, 即记录区间插入哪些元素, 这样元素就被存储在 $O(\lg n)$ 个节点中。处理完所有元素后 DFS 构建数据结构, 因为子区间的数据结构可以继承自父区间的数据结构。注意叶子节点的数据结构不能同时存储, 我们每次只要处理当前 DFS 的链。

该方法参考了 liu_runda 的博客[6]。

3.3 划分树

划分树是一种类似于线段树但很少使用的数据结构, 用来求解静态区间第 k 大问题。划分树可用主席树代替, 权当了解。

[6]bzoj4184shallot
https://www.cnblogs.com/liu-runda/p/6705583.html?utm_source=itdadao&utm_medium=referral

3.3.1 构建

和线段树类似, 在下一层中将每一段区间分为两个子区间, 以这段区间的中位数为划分依据, 并且被划分到同一边的数的相对次序不变。为了支持查询, 还需要记录区间内每一个数及之前的数有多少个划分到左区间中。

3.3.1.1 注意事项

- 由于每一层都只有 n 个数, 所以只要每一层开 n 个数的数组。记 $A[d]$ 为第 d 层的划分情况, $cnt[d][i]$ 为第 d 层的第 i 个数及同区间之前的数有多少个数进入了左子树。
- 中位数可预先对数组排序获得, 记排序后数组为 B。
- 注意有多个中位数的情况(否则左区间的数将覆盖到右区间的存储区域上)。

3.3.2 查询

1. 当到达叶子节点时直接返回 $A[d][i]$ 或 $B[i]$。
2. 利用 cnt 数组差分可以查询到 $[l, r]$ 之间有多少数进入了左子树, 决定往哪棵子树走。
3. 根据 cnt 数组重新计算目标范围在下一层的区间, 递归查询。

3.3.3 板子

SP3946 MKTHNUM - K-th Number

PartitionedTree

```
0 #include <algorithm>
  #include <cstdio>
  #include <cstring>
  int read() {
      int res = 0, c;
      bool flag = false;
      do {
          c = getchar();
          flag |= c == '-';
      } while(c < '0' || c > '9');
10    while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
          c = getchar();
      }
      return flag ? -res : res;
  }
  const int size = 100005;
  int B[size], A[20][size], cnt[20][size];
  void build(int l, int r, int d) {
      if(l == r)
20        return;
      int m = (l + r) >> 1, lp = l, rp = m + 1,
          key = B[m], *AA = A[d], *NAA = A[d + 1],
          *AC = cnt[d], lts = m - l + 1;
```

```
    for(int i = l; i <= r; ++i)
        if(AA[i] < key)
            --lts;
    for(int i = l; i <= r; ++i) {
        AC[i] = (i == l ? 0 : AC[i - 1]);
        int val = AA[i];
        if(val < key || (val == key && lts > 0)) {
            NAA[lp++] = val;
            ++AC[i];
            if(val == key)
                --lts;
        } else
            NAA[rp++] = val;
    }
    build(l, m, d + 1);
    build(m + 1, r, d + 1);
}
int query(int l, int r, int d, int nl, int nr, int k) {
    if(l == r)
        return A[d][l];
    int cl = l == nl ? 0 : cnt[d][nl - 1],
        cr = cnt[d][nr], delta = cr - cl,
        m = (l + r) >> 1;
    if(k <= delta) {
        int lb = l + (cl + 1) - 1;
        int le = l + cr - 1;
        return query(l, m, d + 1, lb, le, k);
    } else {
        int rb = (m + 1) + ((nl - l) - cl);
        int re = rb + ((nr - nl + 1) - delta) - 1;
        return query(m + 1, r, d + 1, rb, re,
                     k - delta);
    }
}
int main() {
    int n = read();
    int q = read();
    for(int i = 1; i <= n; ++i)
        A[0][i] = read();
    memcpy(B + 1, A[0] + 1, sizeof(int) * n);
    std::sort(B + 1, B + n + 1);
    build(1, n, 0);
    while(q--) {
        int l = read();
        int r = read();
        int k = read();
        printf("%d\n", query(1, n, 0, l, r, k));
    }
    return 0;
}
```

以上内容参考了 hchlqlz[7]的博客。

3.4 二叉搜索树

3.4.1 二叉搜索树性质

性质 3.1 向二叉搜索树中插入新节点 u(节点关键字不重复),该节点要么是其前驱的右儿子,要么是其后继的左儿子。

证明 u 的前驱后继在插入节点 u 前互为前驱后继, 若前驱有右儿子则说明前驱比后继高, 后继没有左儿子, 节点 u 成为后继的左儿子。反之亦然。

3.4.2 FHQTreap

FHQTreap 是一种比较好写的二叉搜索树, 虽然效率不太高(不如子节 3.4.3的 splay), 但其易于理解, 不需要旋转, 且对可持久化友好。FHQTreap 的实现基于两个操作:split 与 merge。

对于二叉搜索树的操作, 由于 FHQTreap 本来就是一棵二叉搜索树, 因此操作方法相同(若对效率要求不高, 可以灵活地使用 split 和 merge 以减少代码量)。

对于序列操作, 使用子节 3.4.2.1所述的 splitKth 完成区间提取。

3.4.2.1 split

split 函数的作用是将一棵树按照权值或位置划分成两棵子树。

- 按位置划分:$split(rt, k, x, y)$ 表示将以 rt 为根的树分为以 x 为根的左子树和以 y 为根的右子树, 其中左子树内的节点是原树的前 k 个, 需要维护每个节点的子树大小 siz。

  代码如下:

splitKth

```
0     void split(int u,int k,int& x,int& y) {
          if(u) {
              push(u);
              int lsiz=T[T[u].ls].siz;
              //lsiz(+1)决定当节点u为第k个时被分到哪棵子树
              if(k<=lsiz) {
                  y=u;
                  split(T[u].ls,k,x,T[u].ls);
              }
              else{
10                 x=u;
                   split(T[u].rs,k-lsiz-1,T[u].rs,y);
              }
              update(u);
          }
          else x=y=0;
      }
```

[7]划分树讲解 - hchlqlz https://www.cnblogs.com/hchlqlz-oj-mrj/p/5744308.html

- 按权值划分：$split(rt, k, x, y)$ 表示将以 rt 为根的树分为以 x 为根的左子树和以 y 为根的右子树，其中左子树内的节点值均小于等于 k。

  代码如下：

splitKey

```
0     void split(int u,int k,int& x,int& y) {
          if(u) {
              push(u);
              //=决定当T[u].val==k时被分到哪棵子树
              if(T[u].val<=k) {
                  x=u;
                  split(T[u].rs,k,T[u].rs,y);
              }
              else{
                  y=u;
10                split(T[u].ls,k,x,T[u].ls);
              }
              update(u);
          }
          else x=y=0;
      }
```

根据树的实际意义（二叉搜索树还是序列）以及实际需要来决定使用哪种 split。

3.4.2.2 merge

merge 将两棵树按照左右顺序（中序遍历）合并。和 treap 一样，merge 使用随机权重来保持树的平衡。

代码如下：

merge

```
0     int merge(int u,int v) {
          if(u && v) {
              if(T[u].pri<T[v].pri) {
                  push(u);
                  T[u].rs=megre(T[u].rs,v);
                  update(u);
                  return u;
              }
              else{
                  push(v);
10                T[v].ls=merge(u,T[v].ls);
                  update(v);
                  return v;
              }
          }
          return u|v;
      }
```

3.4.2.3 指示权重的伪随机数生成器

FHQTreap 是一个 Treap, 所以需要一个表现良好的伪随机数生成器来指示节点的权重。

最易于实现的伪随机数生成算法就是线性同余法 (LCG) 了。C++11 中 <random> 的 $std::linear\_congruential\_engine$ 给出了两组预置的参数:

- $minstd\_rand0 : (a = 16807, c = 0, m = 2147483647)$

  Discovered in 1969 by Lewis, Goodman and Miller, adopted as "Minimal standard" in 1988 by Park and Miller.

- $minstd\_rand : (a = 48271, c = 0, m = 2147483647)$

  Newer "Minimum standard", recommended by Park, Miller, and Stockmeyer in 1993.

通常选择第 2 个即 $a = 48271$, 代码如下:

minstd_rand

```
int getRand() {
    static int seed = 347;
    return seed = seed * 48271LL % 2147483647;
}
```

现在尝试说明它的优越性:

- 根据节 4.11所述, 如果 g 是模数 P 的一个原根, 则 g 的幂模 P 可以取到 $[1, P-1]$ 内的每一个数, 且循环周期长度为 $P-1$。由于 2147483647 是梅森素数, 所以它一定存在原根。下列程序可证明 48271 是 2147483647 的一个原根:

RandomTestA.cpp

```
#include <cstdio>
const int mod = 2147483647, p = mod - 1;
typedef long long Int64;
Int64 powm(Int64 a, int k) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
bool test(int g) {
    int x = p;
    for(int i = 2; i * i <= x; ++i)
        if(x % i == 0) {
            do
                x /= i;
            while(x % i == 0);
            if(powm(g, p / i) == 1)
                return false;
```

```
            }
        if(x != 1)
            return powm(g, p / x) != 1;
        return true;
    }
    int main() {
        int g;
        scanf("%d", &g);
        puts(test(g) ? "Yes" : "No");
        return 0;
    }
```

- 48271 是 2147483647 的一个大原根, 可以较早地使 int 溢出, 从而避免出现因 a 过小而导致产生 “锯齿波”。

  下列程序证明了 48271 可以满足 OI 考试的需要:程序输出 minc=7884 maxc=8515 except=8192 s2=88.492, 可见该算法生成的随机数还是蛮均匀的。

RandomTestB.cpp

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <limits>
int getRand() {
    static int seed = 347;
    return seed = seed * 48271LL % 2147483647;
}
const int size = 1 << 23, pool = 1 << 10;
int cnt[pool];
int main() {
    for(int i = 1; i <= size; ++i)
        ++cnt[getRand() >> 21];
    int minv = 1 << 30, maxv = 0, avg = size / pool;
    double s2 = 0.0;
    for(int i = 0; i < pool; ++i) {
        minv = std::min(minv, cnt[i]);
        maxv = std::max(maxv, cnt[i]);
        double delta = cnt[i] - avg;
        s2 += delta * delta;
    }
    s2 /= pool;
    printf("minc=%d maxc=%d except=%d s2=%.3lf\n",
           minv, maxv, avg, sqrt(s2));
    return 0;
}
```

- 模数为 2147483647, 随机数值域广。

  参见 cppreference[8]与 Wikipedia-EN[9]。

[8] https://en.cppreference.com/w/cpp/numeric/random/linear_congruential_engine
[9] https://en.wikipedia.org/wiki/Linear_congruential_generator

如果需要更均匀的随机数, 可以使用如下方案(质量从低到高):

1. 使用比 LCG 更好的梅森旋转算法 [10];
2. RDRAND 指令与 $std :: random\_device$;
3. 使用由一些机构提供的真随机数生成器 SDK, 如https://www.random.org/;
4. 在需要蒙特卡洛采样的场合使用低差异序列如 Halton, Sobol 等。

3.4.2.4 用于可持久化时的注意事项

在可持久化操作中涉及复制整棵子树, 使其子树的随机优先级与原来相同, 形态保持不变, 容易导致后续操作的树不再平衡。

简单的解决方法是不存储随机优先级, 而是维护子树大小, 根据两棵子树的大小~~建立离散分布~~随机决定哪个节点在上方。

该方法来自 IOI2018 国家候选队论文集董炜隽的论文《浅谈 Splay 和 Treap 的性质及其应用》。

3.4.2.5 Finger Search

Finger Search 的思路就是如果操作元素有序, 那么可以从上一次操作位置开始移动, 可以使时间复杂度降一个 lg。

splay 原生支持 Finger Search, 但是 FHQTreap 需要一些额外代码。

具体做法是给每个节点一个管辖区间, 从元素 x 移动到 y 时, 若 y 在管辖区间中则递归下去, 否则向上跳到父亲直至 LCA 处。

该方法来自 IOI2018 国家候选队论文集董炜隽的论文《浅谈 Splay 和 Treap 的性质及其应用》。

3.4.3 splay

由于 Treap 做 LCT 复杂度多一个 log(而且我看不懂), 常数较大, splay 仍然无法被完全替代。

splay 主要由 *rotate* 和 *splay* 函数组成:

3.4.3.1 rotate

$rotate(u)$ 表示将节点 u 旋转到 u 的父亲上。

主要思想是提取出父亲所在的子树, 然后把节点 u 提为子树的根, 最后把一个原来的儿子挂到原来的父亲下以保持二叉平衡树的性质。

具体步骤:

1. 把自己挂到父亲的位置上;
2. 根据相对位置把父亲挂到自己下方;
3. 把父亲占用位置所在的节点(原来的儿子)挂到原来自己的位置上;
4. 依次更新原父亲与自己的信息。

[10] std::mersenne_twister_engine - cppreference.com https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine

rotate 前需要提前将标记下传（在 *splay* 中 *push*）。在实践中可使用辅助函数 $connect(u,p,c)$ 把节点 u 挂到节点 p 的位置 c 下，$getPos(u)$ 获得节点 u 相对于父亲的位置。

代码如下：

rotate

```
int getPos(int u) {
    return u == T[T[u].p].c[1];
}
void connect(int u, int p, int c) {
    T[u].p = p;
    T[p].c[c] = u;
}
void rotate(int u) {
    int ku = getPos(u);
    int p = T[u].p;
    int kp = getPos(p);
    int pp = T[p].p;
    int t = T[u].c[ku ^ 1];
    connect(u, pp, kp);
    connect(t, p, ku);
    connect(p, u, ku ^ 1);
    update(p);
    update(u);
}
```

3.4.3.2 splay

$splay(u,p)$ 表示将节点 u 旋转到 p 的儿子的位置上。*splay* 操作有单旋与双旋之分，其中单旋会被卡(原因见下面所述)。

单旋 单旋的思路很简单，就是不断地把自己旋转到父亲的位置上。

但是当原树是一条链时，单旋 *splay* 后仍然是一条链，树的高度并没有得到优化，因此可以构造数据使得每次 *splay* 都达到最坏时间复杂度 $O(n)$。

双旋 当自己，父亲，爷爷在同一条直线上时，会多旋转一次父亲来尽可能地减小树高。这种做法在链上表现良好。

代码如下：

splay

```
void splay(int u,int p) {
    //push p->u
    while(T[u].p!=p) {
        int pu=T[u].p;
        if(T[pu].p!=p)rotate(getPos(u)==getPos(pu)?pu:u);
        rotate(u);
    }
}
```

3.4.3.3 具体应用

树根 显式维护树根是一件麻烦事, 容易发现 $getPos(root)$ 总是返回 0, 所以 $T[0].c[0]$ 即为 $root$。

二叉搜索树 对于二叉搜索树的操作, 同子节 3.4.2相同, 直接当做二叉搜索树进行操作。为了优化树高(splay 的复杂度是均摊 $\lg n$ 而不是严格的), 每次执行完操作后对最近访问节点 u 调用 $splay(u, 0)$**(血泪史:[HNOI2002] 营业额统计在 insert 重复时也要 splay)**。

序列操作 对于序列操作, 可使用 splay 来提取区间。如果需要操作区间 $[l, r]$, 则:

1. 将节点 $l-1$$splay$ 到根节点;
2. 将节点 $r+1$$splay$ 到 $l-1$ 的右儿子上;
3. 此时 $r+1$ 的左子树就是目标区间。

为了操作区间 $[1, n]$, 可以在头尾分别添加一个哨兵。

splay 合并 $join(T1, T2)$ 表示将树 $T1$ 与 $T2$ 合并为一棵树, 其中 $T1$ 的所有元素权值均小于 $T2$ 的任意元素权值。

做法很简单:把 $T1$ 的最大节点 $splay$ 到根, 此时该节点没有右子树, 然后把 $T2$ 的根挂到该节点右儿子的位置上。

上述内容参考了自为风月马前卒的博客[11]。

splay 启发式合并 维护 splay 树的 size, 合并时将 size 小的拆开插入 size 大的, 时间复杂度 $O(n \lg^2 n)$。据说拆树时使用中序遍历, 插入复杂度均摊为 $O(1)$, 时间复杂度 $O(n \lg n)$。

该说法是有依据的, 参见 Cppreference[12]中带 hint 插入元素的复杂度说明。

注意 FHQTreap 的 merge 必须保证两树无交且参数顺序按照合并后的序列顺序, 不能简单地将两树的根 merge。

3.5 动态树

警告:首先考虑使用 DFS 序解决!

3.5.1 常规操作

动态树是一堆 splay 组成的森林, 主要以 $access$ 和 $makeRoot$ 操作为基础, 可以实现 $link, cut, split, find$ 等功能。LCT 的 splay 用来维护以深度为关键字的重链信息。

3.5.1.1 splay 部分

与常规 splay 的不同之处在于 LCT 中的 splay 的根也是有父亲的, 指向另一棵 splay 的节点, 在此使用辅助函数 $isRoot(u)$ 来判断节点 u 是不是 splay 的根。为了实现 LCT 的 $access$ 功能, 需要支持区间翻转。

[11] splay 详解(一)- 自为风月马前卒 http://www.cnblogs.com/zwfymqz/p/7896036.html
[12] std::set::insert - cppreference.com https://en.cppreference.com/w/cpp/container/set/insert

splay

```
int getPos(int u) {
    return u == T[T[u].p].c[1];
}
bool isRoot(int u) {
    int p = T[u].p;
    return T[p].c[0] != u && T[p].c[1] != u;
}
#define ls T[u].c[0]
#define rs T[u].c[1]
void pushDown(int u) {
    if (T[u].rev) {
        std::swap(ls, rs);
        T[ls].rev ^= 1;
        T[rs].rev ^= 1;
        T[u].rev = false;
    }
}
void update(int u);
void connect(int u, int p, int c) {
    T[u].p = p;
    T[p].c[c] = u;
}
void rotate(int u) {
    int ku = getPos(u);
    int p = T[u].p;
    int kp = getPos(p);
    int pp = T[p].p;
    int t = T[u].c[ku ^ 1];
    T[u].p = pp;
    if (!isRoot(p))
        T[pp].c[kp] = u;
    connect(t, p, ku);
    connect(p, u, ku ^ 1);
    update(p);
    update(u);
}
void push(int u) {
    if (!isRoot(u)) push(T[u].p);
    pushDown(u);
}
void splay(int u) {
    push(u);
    while (!isRoot(u)) {
        int p = T[u].p;
        if (!isRoot(p))
            rotate((getPos(u) == getPos(p)) ? p : u);
        rotate(u);
    }
```

```
}
```

3.5.1.2 access

$access(u)$ 的功能是使节点 u 与其 LCT 的根在同一棵 $splay$ 内, 此时节点 u 是根节点重链中深度最大的点(已经和原来到儿子的边断开)。

具体操作如下:

1. 将节点 u 翻转到其所在 $splay$ 的根;
2. 删除节点 u 的右子树, 即断开与重儿子的连边, 更新节点 u;
3. 跳到节点 u 的父亲节点, 并翻转其至根;
4. 把以 u 为根的链挂到父节点的右子树位置上, 该操作自动断开父节点到原来重儿子的连边, 更新;
5. 重复步骤 3 合并重链直至合并到根节点为止。

代码:

access

```
0 void access(int u) {
      int v = 0;
      do {
          splay(u);
          rs = v;
          update(u);
          v = u;
          u = T[u].p;
      } while(u);
  }
```

3.5.1.3 makeRoot

$makeRoot(u)$ 的功能是把节点 u 翻转为整棵 LCT 的根。

具体操作如下:

1. 打通节点 u 到根节点的路径, 并使节点 u 成为 splay 的根;
2. 由于节点 u 是 splay 中深度最大的节点, 翻转整棵 splay 后就可以使节点 u 成为根了。

代码:

makeRoot

```
0 void makeRoot(int u) {
      access(u);
      splay(u);
      T[u].rev ^= 1;
      pushDown(u);
  }
```

3.5.1.4 split

若要提取 $u-v$ 的路径,$makeRoot(u), access(v), splay(v)$ 后,节点 v 的子树就保存了 $u-v$ 的路径信息。

3.5.1.5 find

首先 $access(u), splay(u)$ 让 u 成为 LCT 根所在的 splay 的根。根节点就是 splay 的最小节点(注意推送翻转标记)。

坑:有些毒瘤出题人可能会卡找最小节点的过程,因此在 find 找到根节点 u 后再执行一次 $splay(u)$。**注意如果 $find$ 中使用了 $splay(u)$ 并且 cut 中使用了 $find$,调用 $find$ 后 u 在根位置,cut 的后续代码需要稍微修改。**

该坑源自 FlashHu 的博客[13]。

3.5.1.6 link

$makeRoot(u)$ 后使 $T[u].p=v$,注意在 link 前要进行连通性检测。

3.5.1.7 cut

- 若保证 u,v 连接,则 split 后令 $T[u].p=T[v].c[0]=0$(节点 v 深度最大,因此只需和左子树断开),注意要更新节点 v。
- 若不保证 u,v 连接,则在 $makeRoot(u)$ 后检查以下条件:
  - 节点 v 所在 LCT 的根是 u;
  - 节点 u 的父亲是 v(查根时执行完 $find(v)$ 后节点 v 已经是节点 u 所在 splay 的根);
  - 节点 v 的左儿子是 u;
  - 节点 u 没有右儿子(若有则说明 $u-v$ 中有其他节点)。

  由于 $find(v)$ 隐式执行了 $access(v), splay(v)$,只需令 $T[u].p=T[v].c[0]=0$。

以上内容参考了 Saramanda 的博客[14]。

3.5.2 技巧/常见方法

3.5.2.1 DSU 优化连通性检测

如果可以保证处于同一连通块内的点不会再次分离(允许临时分离,比如动态 MST 在环上换边时),可以使用本章 3.6节所述的并查集代替 3.5.1.5中昂贵的 $find$。

3.5.2.2 使用 access 进行从某节点到根的路径的染色

$access(u)$ 后节点 u 到根节点的路径上的点在同一棵 splay 内,可用于模拟染色过程。

参见 [SDOI2017] 树点涂色[15]。

---

[13]LCT 总结——概念篇 + 洛谷 P3690[模板]Link Cut Tree(动态树)(LCT, Splay)https://www.cnblogs.com/flashhu/p/8324551.html

[14]LCT (Link-Cut Tree) 详解 (蒟蒻自留地) https://blog.csdn.net/saramanda/article/details/55253627

[15]【P3703】[SDOI2017] 树点涂色 - 洛谷 https://www.luogu.org/problemnew/show/P3703

3.5.2.3 动态 LCA

注意 link/cut 操作不能换根, 换根会使树的形态改变导致 LCA 不同。

求节点 u, v 的 LCA 时, 首先 $access(u)$ 开辟一条链, 类似地 $access(v)$ 再开辟一条链, $access(v)$ 时会与 $access(u)$ 的链相交, 交点即为 LCA。在实现时可以直接令 $access$ 函数返回 v 值作为与上一条链的交点(当 $while(u)$ 为 false 时, 说明在 $u = T[u].p$ 之前的 u 是链的交汇点, 然而过程退出前将 u 赋给了 v)。

板子:SP8791 DYNALCA

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
int getOp() {
    int c, res;
    do
        c = getchar();
    while(c < 'a' || c > 'z');
    while('a' <= c && c <= 'z') {
        res = c;
        c = getchar();
    }
    return res;
}
const int size = 100005;
struct Node {
    int p, c[2];
} T[size];
#define ls T[u].c[0]
#define rs T[u].c[1]
int getPos(int u) {
    int p = T[u].p;
    return u == T[p].c[1];
}
bool isRoot(int u) {
    int p = T[u].p;
    return T[p].c[0] != u && T[p].c[1] != u;
}
void connect(int u, int p, int c) {
    T[p].c[c] = u;
    T[u].p = p;
}
void rotate(int u) {
```

```
    int ku = getPos(u);
    int p = T[u].p;
    int kp = getPos(p);
    int pp = T[p].p;
    int t = T[u].c[ku ^ 1];
    if(isRoot(p))
        T[u].p = pp;
    else
        connect(u, pp, kp);
    connect(p, u, ku ^ 1);
    connect(t, p, ku);
}
void splay(int u) {
    while(!isRoot(u)) {
        int p = T[u].p;
        if(!isRoot(p))
            rotate(getPos(p) == getPos(u) ? p : u);
        rotate(u);
    }
}
int access(int u) {
    int v = 0;
    do {
        splay(u);
        rs = v;
        v = u;
        u = T[u].p;
    } while(u);
    return v;
}
int main() {
    int n = read();
    int m = read();
    while(m--)
        switch(getOp()) {
            case 'k': {
                int u = read();
                int v = read();
                access(u);
                splay(u);
                T[u].p = v;
            } break;
            case 't': {
                int u = read();
                access(u);
                splay(u);
                T[ls].p = 0;
                ls = 0;
            } break;
            case 'a': {
```

```
                access(read());
                printf("%d\n", access(read()));
            } break;
        }
    return 0;
}
```

事实上该方法的适用范围还可以再扩展。考虑每次 $access$ 后将提取出的链打一个标记, 那么在下次 $access$ 时, 每个 u 都意味着这可能是与之前某个节点的 LCA。检查标记可以得到对应的节点标号 (一般结合题目性质贪心保留某个特殊节点)。例题:「雅礼集训 2017 Day7」事情的相似度。

3.5.2.4 LCT 维护边权信息

把边当做节点, 与两端点相连, 端点点权为 0。边的节点编号最好为边编号 + 总端点数, 以便于快速得到原边编号并断开端点连接。

3.5.2.5 LCT 维护子树信息

该方法主要用于解决**不断换根且子树无修改**的问题。若子树有修改则用 TopTree 向轻儿子下放标记。

普通 LCT 只维护了一条链的信息, 即 splay 内信息。维护子树信息的关键在于显式地将虚子树(虚边儿子的子树)信息累加入自身信息中。每个节点维护 2 个数据, 一个是虚子树贡献, 另一个是贡献总和。update 时顺带累加虚子树贡献。

考虑哪些操作会改变虚实子树:

- access:在 $rs = v$ 处将虚子树贡献中 v 的贡献改为 rs 的贡献。
- makeRoot、find、split 等变换树的形态的操作除 access 部分外无影响, 无需特殊处理。
- link:令 $T[u].p = v$ 会导致 v 及其祖先都需要更新贡献, 可以 "$split(u, v)$" 后将 v 换到链顶后把 u 的贡献累加到 v 的虚子树贡献中, 最后执行 $T[u].p = v, update(v)$。如此只影响到了 v 的信息。
- cut:u, v 在同一棵 splay 内, 除 access 部分外无影响。

注意节点与虚儿子一定连接, 但不一定与实儿子连接。对于简单的情况不必考虑这个事实, 比如「BJOI2014」大融合中的子树节点数。但像 LOJ#558「Antileaf's Round」我们的 CPU 遭到攻击 这种题, 子树黑点数的变化会导致子树黑点距离和发生变化, 因为实儿子不一定与自己相连。

接下来以此题为例讨论解决方法, 下文的 "链" 指代 splay 序列:

在查询时需要 $access(u), splay(u)$, 此时 u 必定在 splay 链尾。虽然自己的实儿子与自己不一定相连, 但是自己的左实儿子和虚儿子连到到链的右端以及右实儿子和虚儿子到链的左端一定经过自己。考虑维护自己的子树到 splay 中**实子树对应的链**的左端/右端的距离和。

- 查询时把 $access(u), splay(u)$ 后 u 自己就是链的右端, 并且 u 在 splay 的根, 子树是完整的, u 到链右端的距离和就是答案。
- 维护子树信息仍然使用上述方法。
- 访问到左右端点的信息时注意提前下传翻转标记, 下传翻转标记时同时把**左右端信息交换**。

血泪史：用 LCT 刚「PA 2017」Banany 点分治题时，由于忘了下传翻转标记而加入错误信息，导致无法删除信息（用 set 维护虚子树信息）。在访问 u 的子节点的信息前一定要记得下传!!! 一般在 access 替入 rs 到链左端的信息与 update 读取实儿子信息中需要使用。不过值得欣慰的是我用 LCT 拿了 LOJ 的 rank1，时间是第二快的 50%，空间是第二小的 30%，代码长度是第二短的 80%。

- 为了编码方便，可以将自身的贡献塞到虚子树贡献中。
- LCT 初始化时使用常规 DFS 处理信息，而不是用 link。

代码如下：

```
#include <algorithm>
#include <cstdio>
#include <map>
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
int getOp() {
    int c;
    do
        c = getchar();
    while(c < 'A' || c > 'Z');
    return c;
}
typedef long long Int64;
#define asInt64 static_cast<Int64>
const int size = 100005;
struct Node {
    Int64 sw, swl, swr, isw;
    int p, c[2], sc, isc, w;
    bool rev, col;
} T[size * 5];
#define ls T[u].c[0]
#define rs T[u].c[1]
bool isRoot(int u) {
    int p = T[u].p;
    return T[p].c[0] != u && T[p].c[1] != u;
}
int getPos(int u) {
    int p = T[u].p;
```

```
    return T[p].c[1] == u;
}
void connect(int u, int p, int c) {
    T[u].p = p;
    T[p].c[c] = u;
}
void pushDown(int u) {
    if(T[u].rev) {
        std::swap(ls, rs);
        std::swap(T[u].swl, T[u].swr);
        T[ls].rev ^= 1;
        T[rs].rev ^= 1;
        T[u].rev = false;
    }
}
void update(int u) {
    pushDown(ls);
    pushDown(rs);
    T[u].sc =
        T[ls].sc + T[rs].sc + T[u].isc + T[u].col;
    T[u].sw = T[ls].sw + T[rs].sw + T[u].w;
    T[u].swl = T[ls].swl + T[rs].swl + T[u].isw +
        (T[ls].sw + T[u].w) *
            (T[u].isc + T[u].col + T[rs].sc);
    T[u].swr = T[ls].swr + T[rs].swr + T[u].isw +
        (T[rs].sw + T[u].w) *
            (T[u].isc + T[u].col + T[ls].sc);
}
void rotate(int u) {
    int ku = getPos(u);
    int p = T[u].p;
    int kp = getPos(p);
    int pp = T[p].p;
    int t = T[u].c[ku ^ 1];
    T[u].p = pp;
    if(!isRoot(p))
        T[pp].c[kp] = u;
    connect(t, p, ku);
    connect(p, u, ku ^ 1);
    update(p);
    update(u);
}
void push(int u) {
    if(!isRoot(u))
        push(T[u].p);
    pushDown(u);
}
void splay(int u) {
    push(u);
    while(!isRoot(u)) {
```

```
        int p = T[u].p;
        if(!isRoot(p))
            rotate(getPos(p) == getPos(u) ? p : u);
        rotate(u);
    }
}
void access(int u) {
    int v = 0;
    do {
        splay(u);
        pushDown(rs);
        T[u].isc += T[rs].sc - T[v].sc;
        T[u].isw += T[rs].swl - T[v].swl;
        rs = v;
        update(u);
        v = u;
        u = T[u].p;
    } while(u);
}
void makeRoot(int u) {
    access(u);
    splay(u);
    T[u].rev ^= 1;
    pushDown(u);
}
void split(int u, int v) {
    makeRoot(u);
    access(v);
    splay(v);
}
void cut(int u, int v) {
    split(u, v);
    T[v].c[0] = T[u].p = 0;
    update(v);
}
void link(int u, int v) {
    split(u, v);
    T[u].p = v;
    T[v].isc += T[u].sc;
    T[v].isw += T[u].swl;
    update(v);
}
std::map<Int64, int> eid;
Int64 encode(Int64 u, Int64 v) {
    if(u < v)
        std::swap(u, v);
    return u << 32 | v;
}
void addEdge(int id) {
    int u = read();
```

```
    int v = read();
    T[id].sw = T[id].w = read();
    link(u, id);
    link(v, id);
    eid[encode(u, v)] = id;
}
int main() {
    int n = read();
    int m = read();
    int k = read();
    int cid = n;
    while(m--)
        addEdge(++cid);
    while(k--)
        switch(getOp()) {
            case 'L':
                addEdge(++cid);
                break;
            case 'C': {
                int u = read();
                int v = read();
                int id = eid[encode(u, v)];
                cut(u, id);
                cut(v, id);
                eid.erase(encode(u, v));
            } break;
            case 'F': {
                int u = read();
                access(u);
                splay(u);
                T[u].col ^= 1;
                update(u);
            } break;
            case 'Q': {
                int u = read();
                access(u);
                splay(u);
                printf("%lld\n", T[u].swr);
            } break;
        }
    return 0;
}
```

3.5.2.6 LCT 维护图上信息

若动态图只加边不删边, 仍旧可以使用 LCT 维护图上信息, 关键是 “可缩点”。

额外维护 2 个 DSU, 一个维护连通性, 另一个维护**双连通性**。当且仅当两个点双连通时才缩为一点, 即第二个 DSU 指示该点的 id。

每次访问父亲时都使用父亲的缩点 id 代替, 其它操作不变。

关键在于 $link$ 操作的处理，$link$ 前若两点未连通，执行普通 $link$ 并标记第一个 DSU。否则 $split(u,v)$ 拆出 $u-v$ 的链，遍历整棵 splay 将其所有点的贡献累加到点 v 上，然后把第二个 DSU 中的 id 置为 v。**注意设置位置是原 id，缩点后 v 的左右儿子要置 0。**

上述内容参考了 neither_nor[16]和 GuessYCB[17]的博客，以及 AntiLeaf's Round 的题解[18]。

3.5.2.7 共价大爷游长沙技巧

例题：UOJ#207. 共价大爷游长沙

需要不断给一棵树换边，判断给定路径是否全部经过某条边。

我原先的想法是每次增删路径时，在端点处打标记，查询时统计两边子树端点数。这个想法是错的：考虑有两条路径都经过一条边，如果将这两条路径的一个端点交换，在 LCT 上的标记还是一样的。

但是这个想法离正解已经很接近了。注意这里的问题是“判定”而不是“统计”，考虑给每条路径随机一个权值，增删路径时在端点处异或该权值，查询任意一端子树异或和，判断其是否为路径异或和。

3.5.2.8 LCT 维护黑白连通块

该需求来自 SP16549 QTREE6 - Query on a tree VI。这题当然也可以使用 LCT 维护子树信息的做法，但是讨论比较多，性能不佳（~~树上单点修改 + 换根统计万金油倒是没错，复杂度保证全部交给 LCT~~）。

考虑维护两个森林，一个森林上只有黑点，另一个森林上只有白点，更改颜色时暴力断开原有连边，连到另一棵森林上，查询时只要询问其所在连通块的大小。大小比连通性更容易维护。

但是如果这是一个菊花图，断开中间的点后，可能会导致 $O(n)$ 的边修改量。那么考虑 DFS 一遍给整棵树定型，每次只修改与父亲的连边。那么就保证了连通块内所有边的儿子都是该颜色的点，只有连通块的深度最浅点不是该颜色的点（为了支持根节点的染色，给根节点一个虚父亲），找到深度最浅的点后返回右子树大小（不能返回子树大小-1 是因为左子树可能有其它连通块）。

因为整棵树被定型，不能使用 makeRoot 操作，需要对 link 和 cut 特殊处理。

- $link(u,f_u)$：f_u 要接受 u 子树大小的虚子树贡献，所以要 $access(f_u),splay(f_u)$；为了将 u 挂在下面，要 $splay(u)$；最后连接 $fa[u]=f_u$、贡献 $isiz[f_u]+=siz[u]$、更新 $update(f_u)$。
- $cut(u,f_u)$：$access(u)$ 后要 $splay(f_u)$ 而不是 $splay(u)$，然后按照普通 cut 操作。若使用 $splay(u)$，f_u 并不一定是 u 的左儿子，不能令 $fa[f_u]=0$，而是要令 $fa[lson[u]]=0$。

该方法参考了 cdsszjj 的博客[19]。

3.5.2.9 LCT 离线维护动态图连通性

给定加边与删边操作序列，期间多次询问两点的连通性。

关键在于当连边后会出现环时如何将其变为树。考虑一种贪心的做法，将环上最早被删除的边删掉，这样做将不会影响到图的连通性。而删边时间可以离线预处理。

[16]LCT 维护子树信息（子树信息 LCT）LCT 维护边权（边权 LCT）知识点讲解 https://blog.csdn.net/neither_nor/article/details/52979425

[17]LCT 模板及套路总结 https://www.cnblogs.com/GuessYCB/p/8330024.html

[18]AntiLeaf's Round 题解 https://loj.ac/article/304

[19]bzoj3637: Query on a tree VI【LCT】 https://blog.csdn.net/cdsszjj/article/details/80332588

该方法参考了国家集训队 2014 论文集黄志翱的《浅谈动态树的相关问题及简单扩展》。

不过使用线段树分治 + 按秩合并并查集也可以解决。

3.5.2.10 链权翻转

给 LCT 的每条链再使用一个 splay 来维护链权。均摊复杂度仍为 $O(\lg n)$。

3.6 并查集

3.6.1 路径压缩

路径压缩的原理很简单, 即把找到的最新的祖先存储下来, 于是该节点的深度被缩为 1。注意路径压缩会使树的形状改变, 若维护的数据与树的形状有关则只能使用 LCT 或本节 3.6.2所述的按秩合并。设 $find$ 次数为 f, 时间复杂度 $O(n + f \cdot (1 + \log_{2+\frac{f}{n}} n))$。

3.6.2 按秩合并

对于每个节点维护秩, 代表该节点高度的上界。合并时按照启发式策略将较小秩的连通块并到较大的连通块。设总操作数为 m, 时间复杂度为 $O(m + n \lg n)$。

3.6.3 复杂度证明

留坑待填, 参见算法导论 [4] 中的 21.3 与 21.4 节。

3.6.4 并查集的分裂

若要将集合中的某个点从原集合剥离, 且不需要可持久化(即不查询历史信息), 则可以考虑 "金蝉脱壳", 即保留原节点不动, 但消除其对集合的影响, 另建新点代表该点。具体步骤为为每个点维护一个 id, 指向当前实际所指向的点, 分裂时先消除原 id 指向的点对原集合的影响, 再创建新的点并更新 id。

3.6.4.1 例题

UVA11987 Almost Union-Find [20]

步骤同上所述, **注意只有同时使用路径压缩和按秩合并才能达到 $O(m\alpha(n))$ 的复杂度**。

代码如下:

UVA11987

```
0 #include <cstdio>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
      while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
          c = getchar();
```

[20]【UVA11987】Almost Union-Find - 洛谷 https://www.luogu.org/problemnew/show/UVA11987

```
    }
    return res;
}
const int size = 100005;
struct Node {
    int cnt, fa, rank;
    long long sum;
} T[size * 2];
int find(int u) {
    return T[u].fa == u ? u : find(T[u].fa);
}
void setP(int u, int p) {
    T[u].fa = p;
    T[p].sum += T[u].sum;
    T[p].cnt += T[u].cnt;
}
void merge(int u, int v) {
    u = find(u), v = find(v);
    if(u != v) {
        if(T[u].rank > T[v].rank) {
            setP(v, u);
        } else {
            setP(u, v);
            if(T[u].rank == T[v].rank)
                ++T[v].rank;
        }
    }
}
int id[size], cnt;
void move(int u, int v) {
    int fu = find(id[u]), fv = find(id[v]);
    if(fu != fv) {
        --T[fu].cnt;
        T[fu].sum -= u;
        id[u] = ++cnt;
        T[cnt].fa = fv;
        ++T[fv].cnt;
        T[fv].sum += u;
    }
}
int main() {
    int n, m;
    while(scanf("%d%d", &n, &m) != EOF) {
        cnt = n;
        for(int i = 1; i <= n; ++i) {
            T[i].cnt = 1;
            T[i].fa = i;
            T[i].rank = 0;
            T[i].sum = i;
            id[i] = i;
```

```
        }
        while(m--) {
            int op = read();
            if(op == 3) {
                int p = find(id[read()]);
                printf("%d %lld\n", T[p].cnt,
                       T[p].sum);
            } else {
                int u = read();
                int v = read();
                if(op == 1)
                    merge(id[u], id[v]);
                else
                    move(u, v);
            }
        }
    }
    return 0;
}
```

3.6.5 并查集重构树

类似于 Kruskal 重构树, 当两个连通块相连时建一个新的父亲节点并连边, 可以发现两个节点第一次被连接的时间点就是它们的 LCA 建立的时间点。

例题 BZOJ 3712: [PA2014]Fiolki 可以发现两种物质在同一瓶内当它们在并查集重构树上的 LCA 建立时。以 LCA 深度为第一关键字, 反应优先级为第二关键字对反应进行排序。按照排序后的顺序模拟反应。

代码:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int maxn = 200005, maxk = 500005;
template <int maxv, int maxe>
struct Graph {
    struct Edge {
        int to, nxt;
    } E[maxe];
```

```
    int last[maxv], cnt;
    Graph() : cnt(0) {}
    void addEdge(int u, int v) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u];
        last[u] = cnt;
    }
};
Graph<maxn * 2, maxn * 2> T;
Graph<maxn, maxk * 2> Q;
int fa[maxn * 2];
int find(int u) {
    return fa[u] ? fa[u] = find(fa[u]) : u;
}
struct Query {
    int u, v, d, xorv;
} A[maxk];
int n, d[maxn * 2];
void DFS(int u) {
    for(int i = T.last[u]; i; i = T.E[i].nxt) {
        int v = T.E[i].to;
        d[v] = d[u] + 1;
        DFS(v);
        fa[v] = u;
    }
    if(u <= n)
        for(int i = Q.last[u]; i; i = Q.E[i].nxt) {
            int qid = Q.E[i].to;
            if(A[qid].d)
                A[qid].d = d[find(A[qid].xorv ^ u)];
            else
                A[qid].d = -1;
        }
}
int g[maxn], id[maxk];
bool cmp(int a, int b) {
    return A[a].d > A[b].d;
}
int main() {
    n = read();
    int m = read();
    int k = read();
    for(int i = 1; i <= n; ++i)
        g[i] = read();
    for(int i = 1; i <= m; ++i) {
        int u = find(read()), v = find(read());
        int id = n + i;
        T.addEdge(id, u);
        T.addEdge(id, v);
        fa[u] = fa[v] = id;
```

```
    }
    for(int i = 1; i <= k; ++i) {
        A[i].u = read();
        A[i].v = read();
        if(find(A[i].u) != find(A[i].v)) {
            --i, --k;
            continue;
        }
        A[i].xorv = A[i].u ^ A[i].v;
        id[i] = i;
        Q.addEdge(A[i].u, i);
        Q.addEdge(A[i].v, i);
    }
    memset(fa + 1, 0, sizeof(int) * (n + m));
    for(int i = n + m; i; --i)
        if(!d[i]) {
            d[i] = 1;
            DFS(i);
        }
    std::stable_sort(id + 1, id + k + 1, cmp);
    long long ans = 0;
    for(int i = 1; i <= k; ++i) {
        int qid = id[i];
        int u = A[qid].u, v = A[qid].v;
        int delta = std::min(g[u], g[v]);
        ans += delta;
        g[u] -= delta, g[v] -= delta;
    }
    printf("%lld\n", ans << 1);
    return 0;
}
```

该内容参考了小蒟蒻 yyb 的博客[21]。

3.7 K-D Tree

K-D Tree 是一棵二叉树, 每一层按照某个轴将本节点辖域内的所有点分为较为均匀的两部分, 该节点自身代表用于划分的中点。查询时依靠剪枝来提高查询速度。

3.7.1 构树

具体步骤如下:

1. 对于当前子空间, 选取一个轴来划分(使用 $std :: nth\_element$)出中点;
2. 将中点存储在当前节点上;
3. 递归建左右子树;
4. 更新子树信息。

[21]【BZOJ3712】Fiolki(并查集重构树)https://www.cnblogs.com/cjyyb/p/9368629.html

$std::nth\_element$ 的复杂度为 $O(n)$, 因此构树的复杂度为 $O(n \lg n)$。

在不需要重复使用轴的情况下(比如不需要插入删除), 可以以随机直线为轴, 实际上就是比较坐标的线性加权和大小。直线的分布约均匀越好。

生成均匀的直线不太好做, 将其转换为生成均匀分布的角度, 然后使用三角函数计算权值。为了提高计算速度, 可以提前打表。

均匀直线的生成可以参考 glm 的 circularRand 函数[22]。

3.7.2 插入

3.7.2.1 离线标记

构树时将所有点加入, 记录每个点对应的 id, 但仅维护初始存在的点的信息。加入点时对该点打标记, 同时自底向上更新信息。这种做法虽然保持了最终树的形态良好, 但是在当前存在的点很少时性能不佳。

3.7.2.2 替罪羊树

注意此时每一个节点的划分轴应该是固定的。当二叉树不平衡时会影响查询复杂度, 采用替罪羊树的平衡策略, 维护每棵子树的 size, 如果 $max(siz_l, siz_r) \geq siz_u \cdot fac$ 则暴力重构子树(注意每次插入后只要找到最高的不平衡子树进行重构), 一般 fac 取 0.75。

不过实践表明每固定修改/查询次数暴力重建整棵树跑得更快(~~面向测试数据编程~~)。

3.7.3 删除

删除节点后的处理方法与插入相同。注意被删除的节点可以 gc。

gc

```
0     std::vector<int> pool;
      int newNode() {
          static int cnt=0;
          int id;
          if(pool.size()) {
              id=pool.back();
              pool.pop_hack();
          }
          else id=++cnt;
          return id;
10    }
      void freeNode(int u) {
          pool.push_back(u);
      }
```

3.7.4 查询

1. 如果整棵子树均不满足要求, 就直接返回;
2. 如果整棵子树均满足要求且没有继续递归的必要, 就记录答案或打标记后直接返回;

[22] glm/random.inl at master ·g-truc/glm ·GitHub https://github.com/g-truc/glm/blob/master/glm/gtc/random.inl

3. 计算当前节点, 更新答案;

4. 递归查询左右子树。

在随机数据下, 查询的时间复杂度是 $O(\lg n)$, 在构造数据下复杂度约是 $O(n^{\frac{d-1}{d}})$。证明待补充。

3.7.5 估值

下列为一些常见估值函数:
因为不同轴上的估值两两独立, 所以可以对每个方向贪心计算后求和。

3.7.5.1 曼哈顿距离最小

$w = \sum_{i=1}^{d} max(mind_i - p_i, 0) + max(p_i - maxd_i, 0)$ 当 p_i 在区域内时估值为 0, 在区域外时估值为到最近一边的值(另一边由于符号为负, 值为 0)。

3.7.5.2 曼哈顿距离最大

$w = \sum_{i=1}^{d} max(abs(mind_i - p_i), abs(maxd_i - p_i))$ 选择距离最大的一边。

3.7.5.3 欧几里得距离最小

$w = \sum_{i=1}^{d} max(mind_i - p_i, p_i - maxd_i, 0)^2$

3.7.5.4 欧几里得距离最大

$w = \sum_{i=1}^{d} max((mind_i - p_i)^2, (p_i - maxd_i)^2)$

3.7.6 技巧

以下三种技巧可兼容使用。

3.7.6.1 全局最优值剪枝

如果仅通过该节点维护的子树信息就可以确定子树内不存在更优解, 搜索该子树已经没有意义了, 直接退出。

3.7.6.2 贪心搜索路径选择

与 Alpha-Beta 剪枝的思路相同。先求出两棵子树的估价函数值, 先进入估价更优的子树搜索, 更有可能在该子树中获得最优值, 然后在另一棵子树上获得更多的剪枝机会。

3.7.6.3 预处理降维

如果插入与查询离线, 则可以对某一维排序, 边插入边查询, 降低 K-D Tree 的查询复杂度。

以上内容参考了 n+e 的课件《K-D Tree 在信息学竞赛中的应用》[5]。

3.8 堆

3.8.1 左偏树

左偏树(Leftist Tree)也是一种二叉堆, 核心操作是 $merge$ 函数, 它可以以 $O(\lg n)$ 合并两棵左偏树。

定义外节点为没有左子树或右子树的节点。对于左偏树的每一个节点, 维护其到子树外节点的最近距离, 其中外节点的 $dist = 0, null$ 的 $dist = -1$(其实没必要太严格, 差不多平衡就够了)。

左偏树具有左偏性质:

性质 3.2 $dist_l \geq dist_r$

由此定义可得到一个推论:

推论 3.3 $dist_u = min(dist_l, dist_r) + 1 = dist_r + 1$

考虑一棵距离为 k 的左偏树的最小节点数, 得到以下定理:

定理 3.4 一棵距离为 k 的左偏树为满二叉树时节点数最少, 有 $2^{k+1} - 1$ 个节点。

由此得到推论 3.5:

推论 3.5 一棵节点数为 n 的左偏树, 距离最大为 $[lg(n + 1) - 1]$。

先给出引理 3.6:

引理 3.6 左偏树的最右链恰好有一个外节点。

证明: 由于左偏树是一棵树, 最右链至少有一个外节点; 若存在两个及以上的外节点, 则对于某个非深度最深的点, 必有右子树(否则链就断了), 却没有左子树(由外节点定义可知), 与性质 3.2矛盾。

由推论 3.5与引理 3.6可得如下定理:

定理 3.7 一棵由 n 个节点组成的左偏树最右链最多有 $[lg(n + 1)]$ 个节点。

以上证明参考了阿波罗 2003 的博客[23]。

我原来的简单理解: 对于左偏树中的每一个节点, 维护其子树高度。每次 $merge$ 时先往右子树塞, 若右子树的深度比左子树的深度更大, 就把左子树换过来塞。以此保证树的高度尽可能小。

$merge(u, v)$ 的操作如下:

1. 如果 u 或 v 有一个为 $null$ 则返回另一个节点;
2. 若 v 应该在 u 的上一层则 $swap(u, v)$;

[23]浅谈左偏树 - 阿波罗 2003 https://www.cnblogs.com/yyf0309/p/LeftistTree.html

3. 递归将节点 v 的子树与 u 的右子树合并;

4. 若 $dist_l < dist_r$ 则 $swap$ 左右子树;

5. 更新节点 u 的距离 $dist_u = dist_r + 1$;

6. 返回该树的根 u。

根据定理 3.7可得 $merge$ 的复杂度为 $O(lgn)$。
代码如下(以大根堆为例):

merge

```
int merge(int u, int v) {
    if (u && v) {
        if (T[u].val < T[v].val)
            std::swap(u, v);
        T[u].r = merge(T[u].r, v);
        if (T[T[u].l].dis < T[T[u].r].dis)
            std::swap(T[u].l, T[u].r);
        T[u].dis = T[T[u].r].dis + 1;
    }
    return u | v;
}
```

3.8.1.1 修改

- 插入节点时, 新建一个只有插入元素的堆, 然后合并两个堆。
- 删除节点时, 合并堆顶左右儿子表示的子堆。

3.8.2 斜堆

斜堆的操作与左偏树差不多, 它们的区别是斜堆不维护到外节点的最近距离, 而是在每一次 $merge$ 时简单地 $swap$ 左右子树。

3.8.3 可删堆

一个简单的方法是使用 $std :: multiset$, 但是其常数很大; 更保险的做法是使用两个优先队列(已加入/已删除)来完成操作:

- 加入时将元素加入 ‘‘已加入堆’’;
- 删除时将元素加入 ‘‘已删除堆’’;
- 取堆顶时, 若两堆堆顶相等则弹出, 直到两堆堆顶不相等, 返回 ‘‘已加入堆” 的堆顶。

3.8.4 配对堆

配对堆的复杂度与常数均比左偏树优秀, 其插入、删除和合并复杂度都是 $O(1)$, 但是不支持可持久化(复杂度是均摊的)。其实际性能比复杂度更优秀的斐波那契堆更好。OI 中一般不需要 decrease-key 操作, 因此配对堆比较好写。

下面的内容假设其为小根堆。

配对堆是一个满足堆性质的多叉树(儿子的权值不小于父亲), 在存储时使用左儿子右兄弟表示法。

```
0 struct Node {
      int val,son,bro;
  } T[size];
```

3.8.4.1 合并

合并时直接将堆顶较大的根节点 v 作为堆顶较大的根节点 u 的 son, 原来的 son 变成 v 的 bro。

```
0 int merge(int u,int v) {
      if(u && v) {
          if(T[u].val>T[v].val)
              std::swap(u,v);
          T[v].bro=T[u].son;
          T[u].son=v;
          return u;
      }
      return u|v;
  }
```

3.8.4.2 删除

删除根节点后配对堆剩下一堆由儿子的子树组成的森林, 需要将这些子树合并。配对堆使用 "配对" 的方法使得删除复杂度达到均摊 $O(\lg n)$: 相邻儿子两两配对合并, 再将这些堆合并。

实践时使用 $mergeBro$ 函数将平行的儿子们合并为一个堆。删除堆顶时调用 $mergeBro(T[u].son)$。

```
0 int mergeBro(int u) {
      if(u && T[u].bro) {
          int a=T[u].bro,b=T[a].bro;
          T[u].bro=T[a].bro=0;
          return merge(merge(u,a),mergeBro(b));
      }
      return u;
  }
```

配对堆参考了 WA 自动机的博客[24]。

[24]配对堆 https://wa-am.com/2018/05/13/

3.9 可持久化数据结构

可持久化数据结构的核心思想是 **Copy On Write**(写时复制), 当一个对象将被改变时, 简单地复制其整体, 未修改的部分仍引用原对象的数据, 达到节省拷贝时间与空间的目的。可持久化还容易支持历史查询操作。

可持久化数据结构有主席树(可持久化线段树), 可持久化可并堆, 可持久化 Trie, 可持久化数组, 可持久化并查集, 可持久化平衡树等。

要注意默认拷贝节点 T[0] 的初始值!!!

3.9.1 主席树

用主席树做的经典模型有:

- 差分
- 对于每一个节点为查询左节点, 维护其右边节点为查询右节点时的答案
- 将某一维离散化后不断插入新元素, 预处理出每一时刻的权值线段树, 以支持后续操作

3.9.1.1 区间覆盖单点查询

需求:UOJ#218 火车管理
区间覆盖操作有两种做法:

- 将被完全覆盖的区间的节点的儿子设为 nil, 查询时返回最深的存在节点。
- 将被完全覆盖的区间的节点的儿子设为自己。

3.9.2 可持久化 Trie

若遇到求区间 xor 最大值之类的问题, 使用可持久化 Trie。对于 and,or 最大值问题, 可以在插入完数后把整棵子树加到另一棵子树上去, 查询时只需考虑一边的子树。

3.9.3 可持久化数组

可持久化数组有两种实现:

- 块状数组
- 主席树

可持久化并查集可使用可持久化数组实现。

3.9.4 优化

3.9.4.1 标记永久化

将对整个区间的操作记录在管理此区间的节点, 标记不下传, 统计时标记参与计算。此法节约了 $push$ 的时间且对可持久化友好。

3.9.4.2 克隆开关

若已知按照原方法有一部分数据不被任何时刻的数据结构引用时，直接在该数据上修改（当然也可以 gc，实现比较麻烦）。可以在操作前设置一个 $enableClone$ 开关，若为 $false$ 则直接返回原节点。

代码如下：

cloneA

```
bool enableClone=true;
int cloneNode(int src) {
    if(enableClone) {
        int id=allocNode();
        T[id]=T[src];
        return id;
    }
    return src;
}
```

对于可持久化并查集，若使用路径压缩优化（实践中不太好用），则不好判断是否需要 $clone$。可以在每个节点上记录其被创建时的时间戳，与当前版本时间戳比较。

代码如下：

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 200005;
struct Seg {
    int l, r, t;
} T[size * 40];
int icnt = 0, timeStamp;
int cloneNode(int u) {
    if(T[u].t == timeStamp)
        return u;
    int id = ++icnt;
    T[id].l = T[u].l, T[id].r = T[u].r,
    T[id].t = timeStamp;
    return id;
}
int build(int l, int r) {
    int id = ++icnt;
    T[id].t = timeStamp;
    if(l != r) {
```

```
        int m = (l + r) >> 1;
        T[id].l = build(l, m);
        T[id].r = build(m + 1, r);
    } else {
        T[id].l = l;
        T[id].r = 1;
    }
    return id;
}
int n, p, v1, v2;
int modifyImpl(int l, int r, int src) {
    int id = cloneNode(src);
    if(l == r)
        T[id].l = v1, T[id].r = v2;
    else {
        int m = (l + r) >> 1;
        if(p <= m)
            T[id].l = modifyImpl(l, m, T[id].l);
        else
            T[id].r = modifyImpl(m + 1, r, T[id].r);
    }
    return id;
}
int modify(int rt, int u, int fu, int rk) {
    p = u, v1 = fu, v2 = rk;
    return modifyImpl(1, n, rt);
}
int query(int l, int r, int id, int p) {
    if(l == r)
        return id;
    else {
        int m = (l + r) >> 1;
        if(p <= m)
            return query(l, m, T[id].l, p);
        return query(m + 1, r, T[id].r, p);
    }
}
int find(int& rt, int x) {
    int fx = T[query(1, n, rt, x)].l;
    if(fx == x)
        return x;
    fx = find(rt, fx);
    rt = modify(rt, x, fx, 0);
    return fx;
}
void merge(int& rt, int u, int v) {
    u = find(rt, u), v = find(rt, v);
    if(u != v) {
        int ru = T[query(1, n, rt, u)].r,
            rv = T[query(1, n, rt, v)].r;
```

```
        if(ru < rv)
            rt = modify(rt, u, v, 0);
        else {
            rt = modify(rt, v, u, 0);
            if(ru == rv)
                rt = modify(rt, u, u, ru + 1);
        }
    }
}
int root[size];
int main() {
    n = read();
    root[0] = build(1, n);
    int m = read();
    for(int i = 1; i <= m; ++i)
        switch(read()) {
            case 1: {
                timeStamp = i;
                root[i] = root[i - 1];
                merge(root[i], read(), read());
                break;
            }
            case 2:
                root[i] = root[read()];
                break;
            case 3: {
                root[i] = root[i - 1];
                puts(find(root[i], read()) ==
                            find(root[i], read()) ?
                        "1" :
                        "0");
            } break;
        }
    return 0;
}
```

此法节约了复制节点时的时间与空间, 但是路径压缩增加了修改的时间和空间, 考场上最好只写按秩合并。

3.10 DLX 舞蹈链

DLX 用来求解精确覆盖问题。

精确覆盖问题 给定一个 01 矩阵, 求使得每一列恰好有 1 个 1 的行集合。

3.10.1 X 算法

X 算法使用递归 + 回溯搜索可行解。
算法步骤如下:

1. 从矩阵中选取一行;
2. 将该行和该行所有 1 对应的列以及与该行冲突的行从矩阵中删除得到一个新矩阵。
3. 若该矩阵为空矩阵, 则跳到步骤 4; 否则递归求解新矩阵的精确覆盖, 若返回 false 则返回步骤 1 选取下一行;
4. 若选取的行全部为 1, 则返回 true, 否则返回 false。

3.10.2 DLX

递归 + 回溯使得存储与维护矩阵既麻烦又费时。Donald E.Knuth 使用双向链表来维护矩阵, 这个数据结构被称为 Dancing Links。它利用了双向链表删除与恢复的方便性。

对于矩阵内的每一个 1(此种矩阵一般为稀疏矩阵), 维护其上下左右元素标号和自身坐标。每个元素既是所属行的链表元素, 又是所属列的链表元素。每个列的链表还有链头 C_i(即 0 行元素), 这些链头又与总链头 $head$ 串在一起, 以便检查覆盖情况。

算法步骤如下:

记标示列链表链头 C 为将元素 C 所在列元素以及这些元素所在行元素删除, 回标 C 为其逆操作。

1. 检查 $head.right$ 是否为自身, 若是则覆盖完毕, 输出答案栈内所有元素, 返回 true;
2. 记 $C = head.right$, 标示 C, 枚举 C 所在链表内的行 D:
   (a) 标示元素 D 所在链表行元素对应列链表链头。
   (b) 将其压入答案栈中;
   (c) 递归求解, 若返回 true 则退出, 否则逆序回标, 枚举下一行。
3. 回标 C, 返回 false。

为了提高搜索效率可以维护每列 1 的个数,每次选取 1 个数最少的列遍历。

板子:

```
#include <cstdio>
int getBit() {
    int c;
    do
        c = getchar();
    while(c < '0' || c > '1');
    return c;
}
const int maxa = 505, size = 5000 + maxa;
struct Node {
    int u, d, l, r, x, y;
} A[size];
int siz[maxa];
void mark(int c) {
    A[A[c].l].r = A[c].r, A[A[c].r].l = A[c].l;
    int c1 = A[c].d;
    while(c1 != c) {
        int c2 = A[c1].r;
        while(c2 != c1) {
            A[A[c2].u].d = A[c2].d;
```

```
            A[A[c2].d].u = A[c2].u;
            --siz[A[c2].x];
            c2 = A[c2].r;
        }
        c1 = A[c1].d;
    }
}
void unmark(int c) {
    int c1 = A[c].u;
    while(c1 != c) {
        int c2 = A[c1].l;
        while(c2 != c1) {
            A[A[c2].u].d = A[A[c2].d].u = c2;
            ++siz[A[c2].x];
            c2 = A[c2].l;
        }
        c1 = A[c1].u;
    }
    A[A[c].l].r = A[A[c].r].l = c;
}
int st[maxa], top = 0;
bool DLX() {
    if(!A[0].r) {
        for(int i = 1; i <= top; ++i)
            printf("%d ", st[i]);
        return true;
    }
    int cc = A[0].r, c, csiz = maxa;
    while(cc) {
        if(csiz > siz[cc])
            c = cc, csiz = siz[cc];
        cc = A[cc].r;
    }
    mark(c);
    int c1 = A[c].d;
    while(c1 != c) {
        int c2 = A[c1].r;
        while(c2 != c1) {
            mark(A[c2].x);
            c2 = A[c2].r;
        }
        st[++top] = A[c1].y;
        if(DLX())
            return true;
        --top;
        c2 = A[c1].l;
        while(c2 != c1) {
            unmark(A[c2].x);
            c2 = A[c2].l;
        }
```

```
        c1 = A[c1].d;
    }
    unmark(c);
    return false;
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    int cnt = m;
    A[0].l = m, A[m].r = 0;
    for(int i = 1; i <= m; ++i) {
        A[i - 1].r = i;
        A[i].l = i - 1;
        A[i].u = A[i].d = i;
    }
    for(int i = 1; i <= n; ++i) {
        int f = 0, l = 0;
        for(int j = 1; j <= m; ++j)
            if(getBit() == '1') {
                ++cnt;
                A[cnt].x = j;
                A[cnt].y = i;

                A[cnt].u = A[j].u;
                A[cnt].d = j;
                A[A[j].u].d = cnt;
                A[j].u = cnt;
                ++siz[j];

                if(f == 0)
                    f = cnt;
                if(l) {
                    A[cnt].l = l;
                    A[l].r = cnt;
                }
                l = cnt;
            }
        if(f)
            A[f].l = l, A[l].r = f;
    }
    if(!DLX())
        puts("No Solution!");
    return 0;
}
```

上述内容参考了万仓一黍的博客[25]。

[25]跳跃的舞者, 舞蹈链 (Dancing Links) 算法——求解精确覆盖问题 http://www.cnblogs.com/grenet/p/3145800.html

3.11 Hash 表

由于 std::unordered_set 并不能满足实际性能需要，在此谈谈用于 OI 的 Hash 表设计。

事实上 std::unordered_set 性能低下也是情有可原的，因为它需要面对的应用场合繁多。但在 OI 应用中，我们能够预知全集的大小，元素一般为整数，且只需要支持插入、查询和清空操作，而通用的 std::unordered_set 并不知道这些信息。

下列方法的性能以 LuoguP3727 曼哈顿计划 E 的评测结果为准（不幸的是 std::unordered_set 光荣地 TLE 了）。

3.11.1 链接法

链接法使用 std::vector<int> 当链表，以整数低位为关键字，查询时使用 std::find 暴力搜索，插入时直接 push_back，清空时将已插入值对应的桶清空。

测试结果：943ms。

3.11.1.1 散列函数设计

一个好的散列函数应该考虑 Key 的所有位，而简单地以低位为关键字是个糟糕的选择。模 2^p 和 2^p-1 都不太好。应该选取不太接近 2 的幂的素数。

将模数从 2^{20} 改为 65537 后测试结果为 364ms，优化效果显著。

3.11.2 双重散列 + 开放寻址法

开放寻址法存放一个较大的表，插入时取得 Key 的散列值 $h(Key,0)$，然后查看该处是否有不同元素，若有则移动到位置 $h(Key,1)$，以此类推，直至找到一个空位或发现已插入为止。查询也是如此。

散列函数使用双重散列法，即 $h(k,i)=(h_1(k)+ih_2(k)) \bmod m$。表的大小 m 一般取素数，$h_1(k)=k \bmod m, h_2(k)=c+k \bmod m', m'<m$。一般取两个插入规模 2 倍以上的相邻素数且 $c=1$。装载因子 α 的散列表的平均期望探查数为 $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$。

测试结果：317ms。

~~2019.3.2：为什么又是 rank2。。。~~

使用这种方法时一定要保证关键字非负，尤其是对字符串使用自然溢出 Hash 时。

3.11.3 完全散列

完全散列适用于静态关键字集合的查询。适用于 BSGS 等场合。

留坑待补。

上述内容参考了算法导论 [4] 第 11 章 散列表。

3.12 珂朵莉树/老司机树

珂朵莉树主要用来解决序列上区间元素的复杂变换问题（比如区间幂等无法整体修改的操作）。

使用它的前置条件有：

- 有区间赋值操作
- 数据随机（区间赋值操作的比例稳定且赋值区间均匀）

其主要思路是用 std::set 维护连续权值相同的块, 执行区间操作时暴力切分维护。由于数据随机与区间赋值操作, set 的大小会快速下降。若有 n 个元素, m 次操作, 时间复杂度趋于 $O(m \lg n)$。

3.12.1 存储

单个节点表示区间内的值都等于它的值。存储在 set 中的节点定义如下:

```
0 struct Node {
      int l;
      mutable int val;
      bool operator<(const Node& rhs) const {
          return l<rhs.l;
      }
  };
  std::set<Node> seq;
```

这里只存储左端点比较方便, 其 l 表示区间为 $[l, l_{nxt})$。初始化时在 $n+1$ 处加一个哨兵。

因为 set 的键值不允许改变, decltype(*std::set<Node>::iterator()) 是 const Node&, 所以要将 val 设为 mutable。

3.12.2 切分

$split(pos)$ 在 pos 处切分, 并返回区间 $[pos,)$ 的迭代器。

查找第一个 l 不小于 pos 的位置, 如果其左端点恰好为 pos, 直接返回。否则 pos 在前一个区间中, 将其切分后返回迭代器。

```
0 typedef std::set<Node>::iterator IterT;
  IterT split(int pos) {
      IterT it=seq.lower_bound(pos);
      if(it->l==pos)
          return it;
      --it;
      return seq.insert(Node(pos,it->val)).first;
  }
```

3.12.3 区间赋值

区间赋值是保证时间复杂度的关键操作。只要切分出区间, 然后删掉该区间, 插入新区间就可以了。

```
0 void assign(int l,int r,int val) {
      IterT beg=split(l),end=split(r+1);
      seq.erase(beg,end);
      seq.insert(Node(l,val));
  }
```

由于 split 中只有插入操作, 迭代器保证有效, 不需要注意 split(l) 和 split(r+1) 的顺序。

3.12.4 应用

和区间赋值一样, $split(l), split(r+1)$ 后对 $[beg, end)$ 的迭代器指向的值执行暴力修改或查询。在查询时有时需要得到区间的大小, 有三种解决方案:

- 遍历时维护当前迭代器 it, 同时维护 nxt。
- 在 $Node$ 中维护 r, split 时删除原区间再加入前半段区间。注意此法要先 $split(r+1)$ 再 $split(l)$。
- 在 $Node$ 中维护 $mutable\ int\ r$。

上述内容参考了 zyhang 的博客[26]。

3.13 ETT

ETT 用来解决动态树结构 + 换根 + 维护子树信息的问题。

3.13.1 欧拉遍历序列

以任意一点为根 DFS, 将每条树边拆为两条有向边, 欧拉遍历路径即遍历路径的有向边序列。

事实上这是一个环, 所以支持换根。

3.13.2 基本操作

使用平衡树来维护遍历序列。

3.13.2.1 link

将两棵树并为一棵树。

3.13.2.2 cut

将一棵树分为两棵树。

该内容参考了国家集训队 2014 论文集黄志翱的《浅谈动态树的相关问题及简单扩展》。

留坑待补, 参见 Memphis 的博客[27]。

[26]珂朵莉树详解 https://www.cnblogs.com/yzhang-rp-inf/p/9443659.html
[27]动态树拓展相关 http://memphis.is-programmer.com/2015/8/7/linkcutmemphis.99293.html

Chapter 4

数论

4.1 辗转相除法 GCD 132
4.1.1 裴蜀定理 132
4.1.2 exgcd 133
4.1.3 位运算 gcd 133
4.1.4 O(n)-O(1)gcd 134
4.2 欧拉定理 137
4.2.1 费马小定理 137
4.2.2 线性推逆元 137
4.2.3 欧拉定理 138
4.2.4 扩展欧拉定理 138
4.3 素性测试 139
4.3.1 Miller Rabin 随机性素性测试 139
4.3.2 Baillie-PSW 素性测试 140
4.4 Pollard Rho 启发式因子分解 146
4.4.1 利用 Birthday Trick 提高效率 146
4.4.2 Floyd 判圈法 147
4.5 RSA 算法 147
4.5.1 原理 147
4.5.2 应用 148
4.5.3 RSA 破解 149
4.6 中国剩余定理 CRT 149
4.6.1 CRT 149
4.6.2 ExCRT 149
4.7 积性函数与线性筛 150
4.7.1 定义 150
4.7.2 常见积性函数 150
4.7.3 线性筛 151
4.7.4 积性函数筛 152
4.7.5 因子分解 153
4.8 狄利克雷卷积,狄利克雷逆与莫比乌斯反演 153
4.8.1 狄利克雷卷积 153
4.8.2 狄利克雷逆 154

4.8.3 莫比乌斯反演 . . . 154
4.8.4 常见技巧 . . . 155
4.9 低于线性时间复杂度筛法 . . . 157
4.9.1 杜教筛 . . . 157
4.9.2 min_25 筛 . . . 160
4.9.3 Powerful Number . . . 164
4.9.4 素数 k 次幂前缀和 . . . 166
4.9.5 约数个数函数前缀和 . . . 166
4.10 离散对数问题 . . . 166
4.10.1 BSGS . . . 166
4.10.2 单有原根模数多询问离散对数问题 . . . 169
4.10.3 Pollard rho 算法 . . . 169
4.10.4 Pohlig-Hellman 算法 . . . 172
4.10.5 对 1 求离散对数 . . . 176
4.11 原根 . . . 176
4.11.1 基本定义与定理 . . . 176
4.11.2 求模 n 的原根 . . . 176
4.11.3 原根的应用 . . . 177
4.12 二次剩余及其扩展 . . . 180
4.12.1 勒让德符号 . . . 180
4.12.2 二次剩余 . . . 181
4.12.3 三次剩余 . . . 182
4.12.4 高次剩余 . . . 183
4.12.5 Tonelli-Shanks 算法 . . . 183
4.13 类欧几里得算法 . . . 183

4.1 辗转相除法 GCD

4.1.1 裴蜀定理

定理 4.1 (Bézout's Theorem) 对于任意 $a,b \in \mathbb{Z}$, 关于 x,y 的线性不定方程 (裴蜀方程) $ax+by=c$ 有无穷多整数解 (x,y) 当且仅当 $(a,b)|c$。特别地, 一定存在 (x,y) 使得 $ax+by=(a,b)$ 成立。

由此可得推论:

推论 4.2 a,b 互质的充要条件是存在整数 (x,y) 使得 $ax+by=1$。

接下来证明一定存在 (x,y) 使得 $ax+by=(a,b)$ 成立:

设 s 是 a 和 b 的线性组合集中的最小正元素, 对于某个整数二元组 (x,y) 有 $ax+by=s$, 令 $q=[a/s], r=a \bmod s=a-q(ax+by)=a(1-qx)+b(-qy)$, 所以 r 也是一个线性组合。因为 s 是线性组合集中的最小正元素, 且 $0 \le r < s$, 所以 $r=0$, 可得 $s|a$。同理 $s|b$, 因此 s 是 a,b 的公约数, 可得 $(a,b) \ge s$。因为 $(a,b)|ax+by$ 且 $s>0$, 所以 $(a,b) \le s$。结合 $(a,b) \ge s$ 与 $(a,b) \le s$ 可得 $s=(a,b)$。

以 $(\frac{b}{(a,b)}, \frac{-a}{(a,b)})$ 为步长给初始解 (x,y) 施加偏移可以得到无穷多整数解。

证明参考了霜刃未曾试的博客[1]与算法导论 [4] 第 31.1 节定理 31.2 的证明。

注意求 $ax \equiv b \pmod n$ 最小非负整数解时的周期是 $abs(n/gcd(n,a))$。

[1]关于裴蜀定理的一些证明
https://blog.csdn.net/discreeter/article/details/69833579

4.1.2 exgcd

由定理 4.1可知一定存在整数解 (x,y) 满足 $ax+by=(a,b)$, 如何构造出一组解呢? $exgcd$(扩展欧几里得算法)可求出一组特殊的整数解。
先上代码:

exgcd

```
0   void exgcd(int a,int b,int& x,int& y,int& d) {
        if(b) {
            exgcd(b,a%b,y,x,d);
            y-=a/b*x;
        }
        else x=1,y=0,d=a;
    }
```

该算法由朴素 gcd 修改而来, 同样讨论两种情况:

- $b=0$ 时, gcd 值为 a, 有 $a=1*a+0*b$。
- $b\neq 0$ 时, 考虑递归计算返回的一组解 (y',x') 满足 $by'+(a-[\frac{a}{b}]b)x'=d$, 该式可变形为 $ax'+b(y'-[\frac{a}{b}]x')=d$, 因此本次递归返回 $(x',y'-[\frac{a}{b}]x')$。

4.1.3 位运算 gcd

本节内容源自算法导论 [4] 思考题 33-1。

4.1.3.1 原理

首先为了避免除法操作, 将辗转相除法改为更相减损术, 然后利用 gcd 函数的以下性质进行优化:

- **性质 4.3** 若 a,b 均为偶数,则 $gcd(a,b)=2*gcd(a/2,b/2)$。
- **性质 4.4** 若 a 是奇数,b 是偶数,则 $gcd(a,b)=gcd(a,b/2)$。
- **性质 4.5** 若 a,b 均为奇数,则 $gcd(a,b)=gcd((a-b)/2,b)$。

特别地, 当 $gcd(x,y)$ 的参数中存在 0, 则返回 $x|y$。
算法步骤如下:

1. 特判 x,y 中存在 0 的情况;
2. 根据性质 4.3, 先消去 a,b 的 2 的幂的公因子, 记录幂次 k;
3. 使 a 变为奇数以便于利用性质 4.4, 同时去除 2 的幂的公因子避免重复计算;
4. 利用性质 4.4使 b 变为奇数;
5. 保持 $a<b$ 以便利用性质 4.5;
6. 利用性质 4.5使 $b'=b-a$;
7. 若此时 b 为 0, 则返回 $a*2^k$, 否则重复第 4 步。

该算法利用性质 4.3与 4.4对更相减损术进行了优化。

4.1.3.2 位扫描优化

若某个数末尾有多个 0, 则可以直接右移多位来代替不断右移 1 位。以下是统计末尾 0 的个数 k 的方法:

- GCC 自带了对位扫描指令的封装, 该系列内置函数以 *__builtin_* 为前缀。*__builtin_ctz* 函数返回了参数的末尾 0 个数。
- Sean Eron Anderson 的 **Bit Twiddling Hacks** [2] 中 Counting consecutive trailing zero bits (or finding bit indices) 一节介绍了计算末尾 0 个数的多种方法, 这里只给出 multiply and lookup 法:

countTZ

```
int countTZ(unsigned int x) {
    static const int LUT[32] = {
        0, 1, 28, 2, 29, 14, 24, 3,
        30, 22, 20, 15, 25, 17, 4, 8,
        31, 27, 13, 23, 21, 19, 16, 7,
        26, 12, 18, 6, 11, 5, 10, 9
    };
    return LUT[(((v & -v) * 0x077CB531U)) >> 27];
}
```

这里的神奇常数 **0x077CB531U** 与 De Bruijn Sequences 有关, 在此不详细介绍。LUT 可预处理得出。

4.1.3.3 实现

gcdX

```
int gcdX(int a,int b) {
    if(a && b) {
        int off=countTZ(a|b);
        a>>=countTZ(a);
        do {
            b>>=countTZ(b);
            if(a>b)std::swap(a,b);
            b-=a;
        }while(b);
        return a<<off;
    }
    return a|b;
}
```

4.1.4 O(n)-O(1)gcd

例题:BZOJ4454 C Language Practice
该算法依赖于这个定理:

[2] http://graphics.stanford.edu/~seander/bithacks.html

定理 4.6 任意正整数 x 都可以分解为 $x_1x_2x_3$ 的形式，其中 x_i 要么 $\leq \sqrt{x}$，要么是素数。

在计算 $gcd(x,y)$ 时，如果我们知道了 x 的分解 $x_1x_2x_3$，那么答案可以表示为 $gcd(x_1,y)gcd(x_2,y')gcd(x_3,y'')$，其中 y' 为 y 消去因子 $gcd(x_1,y)$ 后的值，y'' 类似。

接下来分类讨论计算（记 $n = max\{x\}$）：

- 若 $x_i \leq \sqrt{n}$，那么可以 $O(n)$ 预处理 $1 \sim \sqrt{n}$ 两两之间的 gcd（利用 $gcd(x,y) = gcd(x-y,y)$ 递推），然后 $O(1)$ 查询 $gcd(x_i, y\%x_i)$。
- 否则 x_i 是素数，判断 x_i 是否整除 y。

至于计算 x 的因式分解，可以使用线性筛筛出最小质因子 d，然后使用 x/d 的因子分解递推出 x 的分解，选取最小的因子乘以 d。

参考代码：

```
#include <cstdio>
namespace IO {
    char in[1 << 22];
    void init() {
        fread(in, 1, sizeof(in), stdin);
    }
    char getc() {
        static char* S = in;
        return *S++;
    }
}
typedef unsigned int U32;
U32 read() {
    U32 res = 0, c;
    do
        c = IO::getc();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = IO::getc();
    }
    return res;
}
const U32 maxn = 1000000, sqn = 1000;
U32 p[maxn + 5], gcdv[sqn + 5][sqn + 5];
bool flag[maxn + 5];
U32& choose(U32& a, U32& b) {
    return a < b ? a : b;
}
struct Div {
    U32 a, b, c;
    void extend(U32 d) {
        choose(a, choose(b, c)) *= d;
    }
} D[maxn + 5];
void pre() {
    U32 pcnt = 0;
```

```
    D[1].a = D[1].b = D[1].c = 1;
    for(U32 i = 2; i <= maxn; ++i) {
        if(!flag[i]) {
            p[++pcnt] = i;
            D[i].a = i, D[i].b = D[i].c = 1;
        }
        for(U32 j = 1; j <= pcnt && i * p[j] <= maxn;
            ++j) {
            U32 val = i * p[j];
            flag[val] = true;
            D[val] = D[i];
            D[val].extend(p[j]);
            if(i % p[j] == 0)
                break;
        }
    }
    for(U32 i = 1; i <= sqn; ++i)
        gcdv[0][i] = gcdv[i][0] = i;
    for(U32 i = 1; i <= sqn; ++i)
        for(U32 j = 1; j <= i; ++j)
            gcdv[i][j] = gcdv[j][i] = gcdv[i - j][j];
}
U32 tmp[4], c;
U32 gcd(U32 a, U32 b) {
    if(a && b) {
        U32 old = b, d;
        for(U32 i = 1; i <= c && b != 1; ++i) {
            if(tmp[i] <= sqn)
                d = gcdv[tmp[i]][b % tmp[i]];
            else
                d = (b % tmp[i] ? 1 : tmp[i]);
            b /= d;
        }
        return old / b;
    }
    return a | b;
}
const U32 maxq = 2005;
U32 A[maxq];
U32 foo() {
    U32 n = read();
    U32 m = read();
    for(U32 i = 0; i < n; ++i)
        A[i] = read();
    U32 res = 0;
    for(U32 i = 0; i < m; ++i) {
        U32 x = read();
        c = 0;
        if(D[x].a != 1)
            tmp[++c] = D[x].a;
```

```
            if(D[x].b != 1)
                tmp[++c] = D[x].b;
            if(D[x].c != 1)
                tmp[++c] = D[x].c;
            for(U32 j = 0; j < n; ++j)
                res += gcd(x, A[j]) ^ i ^ j;
        }
        return res;
    }
    int main() {
        IO::init();
        pre();
        U32 t = read();
        for(U32 i = 1; i <= t; ++i)
            printf("%u\n", foo());
        return 0;
    }
```

该方法参考了 wzj[3]和 fjzzq2002[4]的博客。

4.2 欧拉定理

4.2.1 费马小定理

定理 4.7 (Fermat's Little Theorem)
$\forall p$ *is a prime number*, $a \in \mathbb{Z}, a^p \equiv a \pmod{p}$

该定理是定理 4.9的特殊化, 不证。
由定理 4.7可得:

推论 4.8 $a^{-1} \equiv a^{p-2} \pmod{p}$

可使用快速幂在 $O(lgp)$ 的复杂度下求某个数的逆元。

4.2.2 线性推逆元

如果需要获得 $a \in [1, p)$ 内模 p 的逆元, 复杂度为 $O(plgp)$ 逐个快速幂的方法并不是最优的。

首先有 $1^{-1} \equiv 1 \pmod{p}$。
令 $p = qa + r$, 其中 $q = [\frac{p}{a}], r = p \bmod a$。
再把 $p \equiv 0 \pmod{q}$ 中的 p 用 $qa+r$ 代替,两边同时乘上 $(ar)^{-1}$,移项得 $a^{-1} \equiv -qr^{-1} \pmod{q}$, 即 $a^{-1} \equiv -[\frac{p}{a}](p \bmod a)^{-1} \pmod{p}$。
代码如下:

inv

```
inv[1]=1;
for(int i=2;i<=n;++i)
    inv[i]=asInt64(mod-mod/i)*inv[mod%i]%mod;
```

[3]BZOJ4454: C Language Practice https://www.cnblogs.com/wzj-is-a-juruo/p/5316963.html
[4]O(1) 查询 gcd https://www.cnblogs.com/zzqsblog/p/5436775.html

以上内容参考了 Miskcoo 的博客[5]

Warning：试验发现这种方法会造成 75% 左右的 Cache Miss，严重影响性能，推荐使用下面的方法，尤其是预处理阶乘逆元时。

Update：还有更一般的做法，可以在 $O(n+\lg p)$ 内推出任意 n 个非 0 数的逆元：

首先计算出前 i 个数的前缀积 M_i，然后快速幂计算 M_n^{-1}，最后从后往前倒推计算每个数的逆元。比如要计算 A_i 的逆元，倒推维护 $X=M_n^{-1}\prod_{j=i+1}^{n}A_j$，那么 $A_i^{-1}=M_{i-1}X$。

该方法源自 WAAutoMaton 的博客(~~知识都是在乱翻他人博客中学到的~~)[6]。

4.2.3 欧拉定理

定理 4.9 (Euler's Theorem)
对于任意互质正整数对 (a,n)，有 $a^{\varphi(n)}\equiv 1 \pmod n$

证明：

令 $S=\{[x]_n\in Z_n|(a,n)=1\}$(由与 n 互质的模 n 剩余类组成的集合)，它与 $\cdot_n$ 构成整数模 n 乘法群，$(S,\cdot_n)$ 的阶为 $\varphi(n)$。

接着有两种证明思路：

- 对于任意一个与 n 互质的正整数 a，a 的幂模 n 的值 $a,a^2,\cdots,a^k$ 构成了一个子群，其中 $a^k\equiv 1 \pmod n$。

  根据定理 5.5，有 $k|\varphi(n)$，令 $M=\varphi(n)/k$，有 $a^{\varphi(n)}=a^{kM}=(a^k)^M\equiv 1^M\equiv 1 \pmod n$。

- 根据定义得对于 $[x]_n\in S$ 和 S 中的所有元素 $[a_1]_n,[a_2]_n,\cdots,[a_{\varphi(n)}]_n$，$[x]_n\cdot_n[a_i]_n$ 组成的集合仍然是 S，因此有 $x^{\varphi(n)}[a_1]_n[a_2]_n\cdots[a_{\varphi(n)}]_n=(x[a_1]_n)(x[a_2]_n)\cdots(x[a_{\varphi(n)}]_n)\equiv[a_1]_n[a_2]_n\cdots[a_{\varphi(n)}]_n \pmod n$，两边消去可得 $x^{\varphi(n)}\equiv 1 \pmod n$。

上述证明源自 Wikipedia-EN[7]和 Eden Harder 的博客[8]。

4.2.4 扩展欧拉定理

定理 4.10 $\forall a\in\mathbb{Z},x,m\in\mathbb{Z}^+,x\geq\varphi(m),a^x\equiv a^{x \bmod \varphi(m)+\varphi(m)} \pmod m$

引理 4.11

$$\begin{cases} x\equiv y \pmod{m_1} \\ x\equiv y \pmod{m_2} \end{cases} \Rightarrow x\equiv y \pmod{lcm(m_1,m_2)}$$

证明：

$$\begin{cases} x\equiv y \pmod{m_1} \\ x\equiv y \pmod{m_2} \end{cases} \Rightarrow \begin{cases} x+c_1m_1=y \\ x+c_2m_2=y \end{cases}$$

$$\Rightarrow c_1m_1=c_2m_2=k\cdot lcm(m_1,m_2)\Rightarrow$$

$$x\equiv y \pmod{lcm(m_1,m_2)}$$

推论 4.12 当 a,b 互质时，$x\equiv y \pmod{ab}$

[5] [数论] 线性求所有逆元的方法–Miskcoo's Space http://blog.miskcoo.com/2014/09/linear-find-all-invert

[6] [loj ???] 乘法逆元 2 题解 https://wa-am.com/2019/03/08/loj-2-

[7] Euler's theorem - Wikipedia https://en.wikipedia.org/wiki/Euler's_theorem

[8] RSA 加密周边 - Eden Harder http://edenharder.is-programmer.com/posts/43247.html

推论 4.13

$$\begin{cases} x \equiv y \pmod{m_1} \\ \cdots \\ x \equiv y \pmod{m_n} \end{cases} \Rightarrow x \equiv y \pmod{lcm(m_1, \cdots, m_n)}$$

引理 4.14

$$\forall p \ is \ a \ prime \ number, q \in \mathbb{Z}^+, q > 1, \varphi(p^q) \geq q.$$

证明:首先有 $\varphi(p^q) = (p-1)p^{q-1}$,当 p 固定时,q 取 2 使得 $\varphi(p^q) - q$ 最小,但该值仍非负。当且仅当 $p = 2, q = 2$ 时,$\varphi(p^q) = q$。

接下来证明定理 4.10:

首先证明当 m 为素数 p 的幂 $(m = p^q)$ 时成立:

- 若 $gcd(a, p) = 1$,则 $gcd(a, p^q) = 1$,根据欧拉定理可证在该情况下成立;
- 若 $gcd(a, p) = p$,由适用范围可知 $x \geq q$,由引理 4.14可知 $x \bmod \varphi(p^q) + \varphi(p^q) \geq q$,因此 $a^x \equiv 0 \equiv a^{x \bmod \varphi(p^q) + \varphi(p^q)} \pmod{p^q}$

对于任意 m,可根据算术基本定理将其分解为素数幂之积。因为 $\varphi(p^q) | \varphi(m)$,所以有 $a^x \equiv a^{x \bmod \varphi(p^q) + \varphi(p^q)} \equiv a^{x \bmod \varphi(m) + \varphi(m)} \pmod{p^q}$。根据引理 4.11及其推论合并这些式子可证明该定理。

以上证明源自后缀自动机·张的文章[9]。

Warning:扩展欧拉定理在模意义下矩阵幂的应用中,有时正确,但是已经出现被 Hack 的例子。尽可能使用矩阵乘法以外的递推方式。

4.3 素性测试

4.3.1 Miller Rabin 随机性素性测试

4.3.1.1 朴素算法

根据 4.2.1中所述的费马定理,若要测试 p 是否为素数,选取基数 $a \in [2, p)$,检查 $a^{p-1} \equiv 1 \pmod{p}$ 是否对所有的 a 均成立。

4.3.1.2 Miller-Rabin

考虑每次随机选取多个 a,当有 k 个 a 满足时,出错的概率最多为 2^{-k}。证明详见算法导论 [4] 第 31.8 节定理 31.39。Miller-Rabin 沿用了朴素算法的思路,并且使用 *witness* 测试来代替朴素检查算法以尽可能避免把 **Carmichael** 数当做素数。

$witness(x, base)$ 返回 *true* 当 *base* 可以证明 x 是合数。

witness

```
0 bool witness(int x, int base) {
      int end = x - 1;
      int c = countTZ(end);
      int t = powm(base, end >> c, x);
      while(c--) {
          int ct = mulm(t, t, x);
          if(t != 1 && t != x - 1 && ct == 1)
```

[9]微小的欧拉定理 EXT 证明 https://zhuanlan.zhihu.com/p/24902174

```
            return true;//case 1
        t = ct;
    }
    return t != 1;//case 2
}
```

接下来证明正确性:

- 如果从 case 1 处返回 *true*, 则说明找到了模 x 意义下 1 的一个非平凡平方根。

  定理 4.15 如果 p 是一个奇素数且 $e \geq 1$, 则方程

  $$x^2 \equiv 1 \pmod{p^e} \tag{4.1}$$

  只有两个解 $x = \pm 1$。

  证明: 方程 4.1等价于 $p^e|(x-1)(x+1)$。因为 $p > 2$, 所以 $p|(x-1)$ 与 $p|(x+1)$ 仅有一个成立(否则 $p|((x+1)-(x-1)) = 2$), 两个解为 $x = \pm 1$。

  推论 4.16 如果模 n 意义下存在 1 的非平凡平方根, 则 n 为合数。

  证明: 定理 4.15的逆否命题也成立, 所以 n 不可能为奇素数的幂, 且对于 $n = 1, 2$ 均不存在非平凡平方根, 因此 n 必为合数。

  根据该推论可得 case 1 有证据证明 x 为合数。

- 如果从 case 2 处返回 *true*, 则说明 x 不满足费马定理。

以上内容参考了算法导论 [4] 第 31.8 节。

4.3.1.3 实现细节

- 在 MillerRabin 之前可先用前几个素数筛掉大部分的合数。
- 如果数据范围在 4759123141 内, 即在 $uint32\_t$ 范围内, 只用 2,7,61 为基数判断。
- 如果数据范围在 10^{16} 内, 使用 2,3,7,61,24251 作为基数, 唯一的强伪素数为 46856248255981。

以上内容参考了 Matrix67 的博客[10]。

4.3.2 Baillie-PSW 素性测试

这个方法在 WC2019 朱震霆的讲课课件《~~简单~~数论算法》中被提及。它在 2^{64} 次方内的结果完全正确, 适用于一般情况。~~现在仍然没有发现确定的合数能够通过这个测试, 不过这样的数确实是存在的~~。试验表明它比 MillerRabin 的效率更低, 代码更长。

该算法结合了基于 2 的强 Fermat 测试 (即 Miller-Rabin 的子过程 witness) 与强 Lucas 测试。虽然这两个测试的伪素数十分多, 但是这两个伪素数集合的交的大小(集合大小仍然是正无穷, 这里的大小可以理解为分布密度)要小得多。~~unsigned long long 内靠谱就行~~。

[10]数论部分第一节: 素数与素性测试 | Matrix67: The Aha Moments http://www.matrix67.com/blog/archives/234

4.3.2.1 强 Lucas 测试

雅可比符号 雅可比符号是勒让德符号 (4.12.1) 的推广。它不再要求 p 是奇素数, 而仅要求 p 是大于 1 的奇数。为了防止误解 p, 这里的 p 使用 n 代替。

雅可比符号具有以下性质:

$$a \equiv b \pmod{n} \quad \Rightarrow \quad \left(\frac{a}{n}\right) = \left(\frac{b}{n}\right) \tag{4.2}$$

$$\left(\frac{a}{n}\right) = \begin{cases} 0 & (a,n) \neq 1 \\ \pm 1 & (a,n) = 1 \end{cases} \tag{4.3}$$

$$\left(\frac{a}{bc}\right) = \left(\frac{a}{b}\right)\left(\frac{a}{c}\right) \tag{4.4}$$

$$\tag{4.5}$$

此外, 二次互反律, 完全积性函数, 以及 $a = 2$ 时的勒让德符号的性质对于雅可比符号仍然成立。

根据性质 4.4可以得出若 $n = \prod p_i^{e_i}$, 则 $\left(\frac{a}{n}\right) = \prod \left(\frac{a}{p_i}\right)^{e_i}$。

雅可比符号的计算可在 $O(\lg a \lg n)$ 的时间内解决, 记过程为 $jacobi(a,b)$:

1. 根据性质 4.2可以等价调用 $jacobi(a\%b, b)$。
2. 根据完全积性函数与 $a = 2$ 时的性质把 a 的因子 2 消去。
3. 若 $b = 1$, 根据完全积性函数的性质, 返回 1。
4. 若 $(a,b) \neq 1$, 根据性质 4.3, 返回 0。
5. 否则 $(a,b) = 1$, 使用二次互反律调用子过程 $jacobi(b,a)$。

这个过程与欧几里得算法很像。

Lucas 序列生成 Lucas 序列有参数 P, Q, D, 可由数列 (U,V) 组合而成, 在此只单独研究这两个数列。

它们满足以下递推关系:

$$\begin{aligned} U_0 &= 0 \\ V_0 &= 2 \\ U_{2k} &= U_k V_k \\ V_{2k} &= V_k^2 - 2Q^k \\ U_{2k+1} &= (PU_{2k} + V_{2k})/2 \\ V_{2k+1} &= (DU_{2k} + PV_{2k})/2 \end{aligned}$$

除法操作中如果分子为奇数则再加上模数???

由递推关系可以从高位到低位构造出 (U_n, V_n)。

判定 若 $U_{n-\left(\frac{D}{n}\right)} \not\equiv 0 \pmod{n}$, 则 n 必为合数。对于 $\left(\frac{D}{n}\right) = -1$, 该条件改为 $U_{n+1} \not\equiv 0 \pmod{n}$。

此外可以检查 $V_{n+1} \not\equiv 2Q \pmod{n}$, 满足任意一个条件就判定它为合数, 这个几乎不增加计算代价的操作提高了判定合数的概率。

4.3.2.2 实现

算法步骤如下:

1. 预筛:使用小素数试除可以筛去绝大多数合数(然而素数判定板子题生成的合数基本上是大质数之积)。
2. 执行基于 2 的强 Fermat 测试。也可以使用其它的基,不过 2 已经经过了大量测试。
3. 从数列 A157142[11] $(1,-3,5,-7,9,-11,13,-15\cdots)$ 的第三项开始,找到第一个整数 D 满足 $\left(\frac{D}{n}\right)=-1$。选择这个数列的原因是易于生成且平均测试次数约为 3.147755149。注意如果 n 是一个完全平方数,那么找不到满足条件的 D,可以使用二分法或牛顿法快速开平方根预先判断。
4. 以参数 $D, P=1, Q=(1-D)/4$ 执行强 Lucas 测试。

模板代码如下:

```
#include <cstdio>
#define PREDIV
typedef long long Int64;
const Int64 maxi = 1LL << 31;
#ifdef ONLINE_JUDGE
typedef __int128 Int128;
Int64 mulm(Int128 a, Int128 b, Int128 mod) {
    return a * b % mod;
}
#else
Int64 addm(Int64 a, Int64 b, Int64 mod) {
    a += b;
    return a < mod ? a : a - mod;
}
Int64 mulm(Int64 a, Int64 k, Int64 mod) {
    Int64 res = 0;
    while(k) {
        if(k & 1)
            res = addm(res, a, mod);
        k >>= 1, a = addm(a, a, mod);
    }
    return res;
}
#endif
Int64 powmBig(Int64 a, Int64 k, Int64 mod) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = mulm(res, a, mod);
        k >>= 1, a = mulm(a, a, mod);
    }
    return res;
}
```

[11] 参见 http://oeis.org/A157142

```
Int64 powm(Int64 a, Int64 k, Int64 mod) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
bool isPerfectSquare(Int64 x) {
    Int64 l = 1, r = 1e9, ans = 0;
    while(l <= r) {
        Int64 m = (l + r) >> 1;
        if(m * m <= x)
            ans = m, l = m + 1;
        else
            r = m - 1;
    }
    return ans * ans == x;
}
int countTZ(Int64 x) {
    return __builtin_ctzl(x);
}
bool Fermat2Big(Int64 x) {
    Int64 d = x - 1, cd = d >> countTZ(d),
          t = powmBig(2, cd, x);
    while(cd != d) {
        Int64 ct = mulm(t, t, x);
        if(ct == 1 && t != 1 && t != d)
            return true;
        cd <<= 1, t = ct;
    }
    return t != 1;
}
bool Fermat2(Int64 x) {
    if(x > maxi)
        return Fermat2Big(x);
    Int64 d = x - 1, cd = d >> countTZ(d),
          t = powm(2, cd, x);
    while(cd != d) {
        Int64 ct = t * t % x;
        if(ct == 1 && t != 1 && t != d)
            return true;
        cd <<= 1, t = ct;
    }
    return t != 1;
}
Int64 gcd(Int64 a, Int64 b) {
    return b ? gcd(b, a % b) : a;
}
```

```
int jacobi(Int64 a, Int64 b) {
    a %= b;
    if(a < 0)
        a += b;

    int x = countTZ(a);
    a >>= x;
    unsigned int modv = b & 7;
    bool flag = (x & 1) && (modv == 3 || modv == 5);

    if(b == 1)
        return flag ? -1 : 1;

    if(gcd(a, b) != 1)
        return 0;

    flag ^= (a & 3) == 3 && (b & 3) == 3;

    int val = jacobi(b, a);
    return flag ? -val : val;
}
Int64 getD(Int64 x) {
    for(Int64 i = 5, add = 2;; add = -add, i = add - i)
        if(jacobi(i, x) == -1)
            return i;
}
struct LucasPair {
    Int64 U, V;
    LucasPair(Int64 U, Int64 V) : U(U), V(V) {}
};
// P=1
LucasPair calcLucasSeqBig(Int64 D, Int64 Q, Int64 mod,
                          Int64 k) {
    Int64 ck = 0, qck = 1;
    LucasPair cp(0, 2);
    for(int i = 62; i >= 0; --i) {
        cp = LucasPair(
            mulm(cp.U, cp.V, mod),
            (mulm(cp.V, cp.V, mod) - 2 * qck) % mod);
        if(cp.V < 0)
            cp.V += mod;
        qck = mulm(qck, qck, mod);
        ck <<= 1;
        if(k & (1LL << i)) {
            Int64 v1 = cp.U + cp.V;
            if(v1 & 1)
                v1 += mod;
            Int64 v2 = mulm(D, cp.U, mod) + cp.V;
            if(v2 & 1)
                v2 += mod;
```

```
            cp = LucasPair((v1 >> 1) % mod,
                           (v2 >> 1) % mod);
            qck = mulm(qck, Q, mod);
            ck |= 1;
        }
    }
    return cp;
}
LucasPair calcLucasSeq(Int64 D, Int64 Q, Int64 mod,
                       Int64 k) {
    D %= mod, Q %= mod;
    if(D < 0)
        D += mod;
    if(Q < 0)
        Q += mod;
    if(mod > maxi)
        return calcLucasSeqBig(D, Q, mod, k);
    Int64 ck = 0, qck = 1;
    LucasPair cp(0, 2);
    for(int i = 62; i >= 0; --i) {
        cp = LucasPair(cp.U * cp.V % mod,
                       (cp.V * cp.V - 2 * qck) % mod);
        if(cp.V < 0)
            cp.V += mod;
        qck = qck * qck % mod;
        ck <<= 1;
        if(k & (1LL << i)) {
            Int64 v1 = cp.U + cp.V;
            if(v1 & 1)
                v1 += mod;
            Int64 v2 = D * cp.U + cp.V;
            if(v2 & 1)
                v2 += mod;
            cp = LucasPair((v1 >> 1) % mod,
                           (v2 >> 1) % mod);
            qck = qck * Q % mod;
            ck |= 1;
        }
    }
    return cp;
}
bool Lucas(Int64 x) {
    Int64 D = getD(x), Q = (1 - D) >> 2;
    LucasPair p = calcLucasSeq(D, Q, x, x + 1);
    return p.U == 0 && p.V == ((2 * Q) % x + x) % x;
}
bool BailliePSW(Int64 x) {
    if(x == 1)
        return false;
#ifdef PREDIV
```

```
    Int64 pa[] = { 2, 3, 5, 7, 11, 13 };  // 80%
    for(Int64 p : pa) {
        if(x == p)
            return true;
        if(x % p == 0)
            return false;
    }
#else
    if(x == 2)
        return true;
#endif
    if(isPerfectSquare(x))
        return false;
    if(Fermat2(x))
        return false;
    return Lucas(x);
}
int main() {
    Int64 x;
    while(scanf("%lld", &x) != EOF)
        puts(BailliePSW(x) ? "Y" : "N");
    return 0;
}
```

正确性证明留坑待补。

上述内容参考了 Wikipedia-EN[12]。英文论文参见 Lucas Pseudoprimes[12] (http://mpqs.free.fr/LucasPseudoprimes.pdf)

4.4 Pollard Rho 启发式因子分解

Pollard Rho 算法可以期望在 $\Theta(\sqrt{p})$ 次算术运算内得到 n 的一个小因子 p。

4.4.1 利用 Birthday Trick 提高效率

4.4.1.1 朴素随机算法

考虑每次随机选择一个数 $x \in [2, n-1]$, 若 $x|n$, 则 x 为 n 的一个因子, 最坏情况下(n 为两素数之积)期望测试次数为 $\frac{n-2}{2}$。

4.4.1.2 Birthday Trick

可以把问题转换为选取 k 个数 x, 每次询问是否存在 $(x_i - x_j)|n$。根据生日悖论, 当 k 达到 $\Theta(\sqrt{n})$ 级别时, 可以期望得到一组 (i, j) 满足该条件。具体证明参见算法导论 [4] 第 5.4.1 节中采用指示器随机变量的分析。

询问是否存在 $(x_i - x_j)|n$ 仍然不够高效,可以转换为询问是否存在 $gcd(x_i - x_j, n) > 1$。只需要选取约 $\sqrt[4]{n}$ 个数(因子大小为 $O(\sqrt{n})$)。

Pollard Rho 采取如下策略:

[12] Jacobi Symbol https://en.wikipedia.org/wiki/Jacobi_symbol
Baillie–PSW primality test https://en.wikipedia.org/wiki/Baillie-PSW_primality_test
Lucas pseudoprime https://en.wikipedia.org/wiki/Lucas_pseudoprime
Lucas sequence https://en.wikipedia.org/wiki/Lucas_sequence

1. 在区间 $[2, n-1]$ 中随机选取 k 个数;
2. 询问是否存在 $gcd(x_i - x_j, n) > 1$。

在实践中把 k 个数存下是不可能的, 可以每次把相邻的随机数当做 x_i, x_j 来测试。有一个简单有效的伪随机数生成函数: $f(x) = (x^2 + c) \bmod n$。

4.4.2 Floyd 判圈法

某些输入可能会导致出现在找到某个因子前, 伪随机数生成过程已掉入死循环的情况。遇到这种情况, 应该及时退出迭代, 并修改随机数种子 x_0 与偏移 c 重新迭代。

如何判断是否掉入死循环呢? 这里使用 Floyd 发明的方法: 令 $A_0 = B_0 = x_0$, 同时迭代生成随机数, $A_{i+1} = f(A_i), B_{i+1} = f(f(B_i))$, 当 $A_i = B_i$ 时, B 至少比 A 多走了一圈, 说明已经掉入了循环。

代码实现:

Pollard Rho

```
int f(int x,int c,int n) {
    return (asInt64(x)*x+c)%n;
}
int pollardRhoImpl(int n,int seed) {
    int c=rand()%n;
    int a=f(seed,c,n),b=f(a,c,n);
    while(a!=b) {
        int d=gcd(iabs(a-b),n);
        if(d!=1 && d!=n)
            return d;
        a=f(a,c,n),b=f(f(b,c,n),c,n);
    }
    return 0;
}
int pollardRho(int x) {
    int d=0;
    do d=pollardRhoImpl(x,rand()%n);
    while(!d);
    return d;
}
```

求 gcd 时可以累乘多次一并求 gcd 以减少常数。

以上内容参考了 **A Quick Tutorial on Pollard's Rho Algorithm** [13]。

4.5 RSA 算法

4.5.1 原理

定理 4.17 (素数定理) $\lim\limits_{n\to\infty} \dfrac{\pi(n)}{n/\ln n} = 1$

[13]原文地址 www.cs.colorado.edu/~srirams/classes/doku.php/pollard_rho_tutorial
中文翻译 http://files.cnblogs.com/files/Doggu/Pollard-rho.pdf

RSA 的实用性与安全性基于以下事实: 寻找大素数很容易(根据定理 4.17, 素数密度足够大。可以随机一个整数然后做素性测试, 期望尝试次数为 $\Theta(\ln n)$), 但把两个大素数之积质因数分解却很难。

RSA 算法的基本步骤如下:

1. 随机选取两个大素数 p, q, 使得 $p \neq q$, 令 $n = pq$;
2. 选取一个与 $\varphi(n) = (p-1)(q-1)$ 互质的小奇数 e, 计算模 $\varphi(n)$ 意义下 e 的乘法逆元 d(由于 e 与 $\varphi(n)$ 互质, 根据定理 4.9得存在唯一解 d);
3. 将 $P(e, n)$ 公开, 作为 **RSA 公钥**;
   将 $S(d, n)$ 保密, 作为 **RSA 私钥**。

对于消息 M, 公钥持有者可进行运算: $P(M) = M^e \bmod n$; 私钥持有者可进行运算: $S(M) = M^d \bmod n$。对于用公/私钥加密 M 得到的密文 C, 只有使用私/公钥解密才能得到 M(加解密操作相同)。由于结果要 $\bmod n$, 消息 M 的域为 Z_n。

下面证明 RSA 算法的正确性, 即:

$$P(S(M)) \equiv S(P(M)) \equiv M^{ed} \equiv M \pmod{n}$$

因为 e, d 是模 $\varphi(n)$ 意义下的乘法逆元, 所以有 $ed = 1 + k(p-1)(q-1)$。

- 若 $M \not\equiv 0 \pmod{p}$, 则有

$$\begin{aligned} M^{ed} &\equiv M^{1+k(p-1)(q-1)} \pmod{p} \\ &\equiv M \cdot (M^{p-1})^{k(q-1)} \pmod{p} \\ &\equiv M \cdot 1^{k(q-1)} \pmod{p} \\ &\equiv M \pmod{p} \end{aligned}$$

- 若 $M \equiv 0 \pmod{p}$, 上述等式仍成立。~~M=0 还有什么加密意义吗~~。。。

同样地, 对于 q 有 $M^{ed} \equiv M \pmod{q}$。根据推论 4.12, 有 $M^{ed} \equiv M \pmod{n}$, 证毕。

实际应用时还需要对密文进行随机填充, 常见的有 Optimal Asymmetric Encryption Padding (OAEP) 算法, 参见标准文档 Public-Key Cryptography Standards(PKCS)。

4.5.2 应用

4.5.2.1 消息加密

发送方使用接收方的公钥 P 把消息 M 加密得到密文 C, 将密文 C 发送给接收方。接收方使用自己的私钥 S 解密得到消息 M。

快速无公钥加密系统 若消息过长, 则仅用 P 加密对称加密算法的随机密钥 K, 同时用密钥 K 加密 M 得到密文 C, 把 $(P(K), C)$ 发送给接收方。接收方使用 S 解密得到 K, 再用 K 对 C 解密。

4.5.2.2 数字签名

发送方使用自己的私钥 S 把消息 M 签署得到签名 C, 将消息 M 与签名 C 发送给接收方。接收方使用发送方的公钥 P 解密 C 得到消息 M, 验证消息是否正确。

快速数字签名 与快速无公钥加密系统类似, 仅签署消息的快速散列函数的值。

证书链 以一个可信根为起点, 大家都信任它并且知道它的公钥。下一级将自己的公钥和由上一级签署的认证信息作为自己的签名证书, 接收方自上而下验证证书链上每一级的正确性, 从而验证证书链尾端消息的正确性。

以上内容参考了算法导论 [4] 第 31.7 节。

4.5.3 RSA 破解

RSA 的安全性基于对大整数做质因数分解的困难性。破解 RSA 的目的是得到密文或签署伪造文件, 这意味着攻击者要得到密钥 S。由于公钥 P 已知, 唯一需要知道的是 d, d 可使用 $\varphi(n)$ 由欧拉定理得到。$\varphi(n)$ 的计算依赖于 n 的质因数分解。

4.6 中国剩余定理 CRT

4.6.1 CRT

定理 4.18 (Chinese Remainder Theorem) 对于模线性方程组:

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \cdots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

其中 $n_1, n_2, \cdots, n_k$ 两两互质, 令 $N = \prod_{i=1}^{k} n_i$, 则该模线性方程组在 $[0, N)$ 内有唯一解。

如何求解该线性方程组呢? 和拉格朗日插值法的思路相同, 记方程 e_i 给最终解贡献 x_i, 满足

$$x_i \bmod n_j = \begin{cases} 0 & \text{if } i \neq j \\ a_i & \text{if } i = j \end{cases}$$

答案即为 $\sum_{i=1}^{k} x_i \bmod N$。

考虑 $i \neq j$ 时 $n_j | x_i$, 因此 x_i 有因子 $M = N/n_i$; 当 $i = j$ 时, x_i 应该有因子 a_i, 为了抵消因子 M 的影响, 再乘上 M 模 n_i 的乘法逆元(由于 n 两两互质, M 与 n_i 也互质, 根据定理 4.9, 保证其乘法逆元存在)。

4.6.2 ExCRT

当 n 不满足两两互质的条件时, 可能会找不到其乘法逆元。所以我们采用另一种思路求解方程: 每次选择两个方程将其合并, 直到只剩一个方程为止。

考虑两个方程组成的方程组:

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \end{cases}$$

等价于

$$x0 = a_1 + k_1 n_1 \tag{4.6}$$

$$x0 = a_2 + k_2 n_2 \tag{4.7}$$

移项得 $k_1 n_1 - k_2 n_2 = a_2 - a_1$, 可以使用 $exgcd$ 求出 $c_1 n_1 + c_2 n_2 = gcd(n_1, n_2)$ 的各项参数。根据定理 4.1, 若 $gcd(n_1, n_2) \nmid (a_2 - a_1)$ 则该方程组无解。等比例缩放方程求出 $k1$, 带入方程 4.6反推出 $x0$, 得到新的模线性方程 $x \equiv x0 \pmod{lcm(n_1, n_2)}$。

4.6.2.1 一般情况

例题:NOI2018 屠龙勇士
模线性方程组具有更一般的形式:$a_i x \equiv b_i \pmod{p_i}$, a, b, p 之间无特殊性质。
解出每个方程的通解, 通解的表示也是模线性方程组, 转化为 ExCRT 解决。
也有不使用 ExCRT 的做法, ~~官方题解就是这样说的~~, ~~但我忘了~~。

4.7 积性函数与线性筛

4.7.1 定义

数论函数(Arithmetic Function) 若函数 $f : \mathbb{Z}^+ \to \mathbb{C}$, 则称函数 f 为数论函数。

积性函数(Multiplicative Function) 若函数 f 为数论函数, 且 $f(1) = 1$, 对于任意互质的正整数 a, b 都有 $f(ab) = f(a)f(b)$, 则称函数 f 为积性函数。

完全积性函数(Completely Multiplicative Function) 若函数 f 为积性函数且对于任意正整数 a, b 都有 $f(ab) = f(a)f(b)$, 则称函数 f 为完全积性函数。

性质 4.19 若 f 为积性函数,对于正整数 $n = \prod_{i=1}^{m} {p_i}^{c_i}$,有 $f(n) = \prod_{i=1}^{m} f({p_i}^{c_i})$

性质 4.20 若 f 为完全积性函数,对于正整数 $n = \prod_{i=1}^{m} {p_i}^{c_i}$,有 $f(n) = \prod_{i=1}^{m} f(p_i)^{c_i}$

4.7.2 常见积性函数

4.7.2.1 积性函数

- 除数函数 $\sigma_k(n) = \sum_{d|n} d^k$,

  根据性质 4.19可得 $\sigma_k(n) = \prod_{i=1}^{m} \sum_{j=0}^{c_i} p_i^{jk}$

- 约数个数函数 $\tau(n) = \sigma_0(n)$
- 约数和函数 $\sigma(n) = \sigma_1(n)$
- 欧拉函数 (Euler Totient Function) $\varphi(n) = \sum_{i=1}^{n} [(n, i) = 1] = n \prod_{p|n} (1 - \frac{1}{p})$, 且有 $\sum_{i=1}^{n} [(n, i) = 1] * i = \frac{n\varphi(n) + [n = 1]}{2}$
- 莫比乌斯函数定义为:

$$\mu(d) = \begin{cases} 1 & \text{if } d = 1 \\ (-1)^k & \text{if } d = \prod_{i=1}^{k} p_i \\ 0 & \text{otherwise} \end{cases}$$

简单来说就是如果存在平方因子则 $\mu(n)$ 为 0, 否则 $\mu(n)=(-1)^k, k$ 为质因子数。

定理 4.21

$$[n=1]=\sum_{d|n}\mu(d)$$

证明: 当 $n=1$ 时, 该等式成立。对于 $n>1$ 的情况, 将 n 分解为 $\prod_{i=1}^{m}{p_i}^{c_i}$, 令 $X=\prod_{i=1}^{m}p_i$, 仅考虑 $\mu(d)\neq 0$ 的部分, $\mu(d)$ 有贡献当且仅当 $d|X$, 因此 d 可表示为一个长度为 m 的 01 向量。由二项式定理可知选取奇数个 1 的向量方案数等于选取偶数个 1 的向量方案数, 即正负贡献抵消。

定理 4.22

$$n=\sum_{d|n}\varphi(d)$$

证明: 将 n 个分数 $\frac{1}{n},\frac{2}{n},\cdots,\frac{n}{n}$ 化为最简分数, $\varphi(x)$ 即表示分母为 x 的最简分数个数。

定理 4.23

$$\begin{aligned}\sum_{i=1}^{n}\tau(i) &= \sum_{i=1}^{n}[\frac{n}{i}]\\ \sum_{i=1}^{n}\sigma(i) &= \sum_{i=1}^{n}i\cdot[\frac{n}{i}]\end{aligned}$$

证明: 枚举因子 i, n 以内有 $[\frac{n}{i}]$ 个因子。

4.7.2.2 完全积性函数

- 元函数(Unit Function) $\epsilon(n)=[n=1]$
- 恒等函数(Constant Function) $1(n)=1$
- 单位函数 $id(n)=n$
- 幂函数 $id^k(n)=n^k$

以上内容参考了 skywalkert 的博客[14]与 Wikipedia-EN[15]。

4.7.3 线性筛

主要思路是每次拿当前的数和已经筛出的素数构造成新的合数并将其筛去。
代码如下:

Euler

```
0 int prime[size/log(size)],pcnt=0;
  bool flag[size];
  void pre(int n) {
      for(int i=1;i<=n,++i) {
```

[14]浅谈一类积性函数的前缀和 https://blog.csdn.net/skywalkert/article/details/50500009
[15]Arithmetic function - Wikipedia https://en.wikipedia.org/wiki/Arithmetic_function

```
        if(!flag[i])
            prime[++pcnt]=i;
        for(int j=1;j<=pcnt && prime[j]*i<=n;++j) {
            flag[prime[j]*i]=true;
            if(i%prime[j]==0)
                break;//case 1
        }
    }
}
```

注意到 case 1 中的优化, 它保证了每个合数最多被筛 1 次, 从而使时间复杂度变为 $O(n)$, 并且增加了一个性质: 合数只被其最小质因子筛去。接下来证明该优化的正确性: 当 $i \bmod p_j = 0$ 时, 有 $i = kp_j$, 若要用 p_{j+x} 筛去后面的合数 $ip_{j+x} = kp_jp_{j+x}$, 可知该合数未来将被合数 kp_{j+x} 与素数 p_j 筛去, 直接跳出不会影响结果, 且保证合数被最小质因子筛除, 便于质因数分解。

如果仅仅要筛 10^6 以内的素数, $O(n \lg \lg n)$ 的埃氏筛法比线性筛更为简洁高效。所以线性筛的主要用途是利用每个合数被其最小质因子筛去的重要性质筛积性函数值。埃氏筛法也可以用枚举倍数的性质使用容斥计算积性函数值。

4.7.4 积性函数筛

4.7.4.1 欧拉函数

- $\varphi(1) = 1$;
- 若 i 为素数, 则 $\varphi(i) = i - 1$;
- 若 $i \bmod p_j = 0$, 则说明 ip_j 存在至少两个因子 p_j, 因此 $\varphi(ip_j) = \varphi(i)p_j$;
- 若 $i \bmod p_j \neq 0$, 则根据积性函数性质可得 $\varphi(ip_j) = \varphi(i)(p_j - 1)$。

4.7.4.2 莫比乌斯函数

- $\mu(1) = 1$;
- 若 i 为素数, 则 $\mu(i) = -1$;
- 若 $i \bmod p_j = 0$, 则说明 ip_j 存在至少两个因子 p_j, 因此 $\mu(ip_j) = 0$。注意若数组已清零则不赋值;
- 若 $i \bmod p_j \neq 0$, 则根据积性函数性质可得 $\mu(ip_j) = -\mu(i)$。

4.7.4.3 约数个数

记数组 A_i 为 i 中最小质因子的次数。

- $\tau(1) = 1, A_1 = 0$;
- 若 i 为素数, 则 $\tau(i) = 2, A_i = 1$;
- 若 $i \bmod p_j = 0$, 则说明 ip_j 存在至少两个因子 p_j, 因此 $\tau(ip_j) = \tau(i) \cdot \frac{A_i+2}{A_i+1}$ 且 $A_{ip_j} = A_i + 1$;
- 若 $i \bmod p_j \neq 0$, 则根据积性函数性质可得 $\tau(ip_j) = 2\tau(i)$ 且 $A_{ip_j} = 1$。

4.7.4.4 约数和

由性质 4.19可得

$$\sigma(n)=\prod_{i=1}^{m}\sum_{j=0}^{c_i}p_i^j$$

记数组 low_i 为 i 中最小质因子的幂, sum_i 为 i 中最小质因子的贡献。

- $\sigma(1)=1, low_1=1, sum_1=1$;
- 若 i 为素数, 则 $\sigma(i)=i+1, low_i=i, sum_i=i+1$;
- 若 $i \bmod p_j=0$, 则说明 ip_j 存在至少两个因子 p_j, 因此 $\sigma(ip_j)=\sigma(i)\cdot\frac{sum_{ip_j}}{sum_i}$ 且 $low_{ip_j}=low_i*p_j, sum_{ip_j}=sum_i+low_{ip_j}$;
- 若 $i \bmod p_j\neq 0$, 则根据积性函数性质可得 $\sigma(ip_j)=(p_j+1)\sigma(i)$ 且 $low_{ip_j}=p_j, sum_{ip_j}=p_j+1$。

4.7.4.5 普通积性函数

同约数和的思想, 记数组 sum_i 为 i 中最小质因子的贡献。要求能够快速推出 $f({p_i}^{c_i})$ 的值。

- $f(1)=1$;
- 若 i 为素数, 则 $f(i)=sum_i=\cdots$;
- 若 $i \bmod p_j=0$, 则说明 ip_j 存在至少两个因子 p_j, 因此 $f(ip_j)=f(i)\cdot\frac{sum_{ip_j}}{sum_i}$;
- 若 $i \bmod p_j\neq 0$, 则根据积性函数性质可得 $f(ip_j)=f(p_j)f(i)$。

以上内容参考了租酥雨的博客[16]。

4.7.5 因子分解

通过在每次筛除时记录其最小质因子, 可以于 $O(\lg n)$ 复杂度内分解因子。

4.8 狄利克雷卷积,狄利克雷逆与莫比乌斯反演

4.8.1 狄利克雷卷积

对于数论函数 f,g, 定义狄利克雷卷积

$$(f*g)(n)=\sum_{d|n}f(d)g(\frac{n}{d})=\sum_{ab=n}f(a)g(b)$$

由积性函数集合与狄利克雷卷积组成的群的乘法单位元为元函数 ϵ。

由定义求 n 以内的狄利克雷卷积的复杂度为 $O(n\lg n)$, 使用调和级数可证明。

狄利克雷卷积有如下性质:

结合律	$(f*g)*h=f*(g*h)$
分配律	$f*(g+h)=f*g+f*h$
交换律	$f*g=g*f$
单位元	$f*\epsilon=\epsilon*f=f$

[16]积性函数与线性筛 - 租酥雨 https://www.cnblogs.com/zhoushuyu/p/8275530.html

4.8.2 狄利克雷逆

已知数论函数 f, 求 $g = f^{-1}$, 满足 $f * g = \epsilon$。

- 当 $n = 1$ 时, 有 $(f * g)(1) = f(1)g(1) = \epsilon(1) = 1$, 解得 $g(1) = \frac{1}{f(1)}$。
- 当 $n > 1$ 时, 有 $(f * g)(n) = \sum_{ab=n} f(a)g(b) = \epsilon(n) = 0$, 解得

$$g(n) = \frac{-1}{f(1)} \sum_{d|n, d<n} f(\frac{n}{d})g(d)$$

4.8.2.1 狄利克雷逆性质

性质 4.24 积性函数的狄利克雷逆仍然是积性函数。

性质 4.25 若数论函数 f, g 为积性函数, 则 $(f * g)^{-1} = f^{-1} * g^{-1}$。

性质 4.26 积性函数 f 为完全积性函数当且仅当 $f^{-1}(n) = \mu(n)f(n)$。

4.8.2.2 常见数论函数及其狄利克雷逆

- $1 * \mu = \epsilon$
  参见定理 4.21的证明。
- $id^\alpha * (\mu \cdot id^\alpha) = \epsilon$
  根据性质 4.26可证明。
- $\varphi * (\sum_{d|n} \mu(d)d) = \epsilon$
  由定理 4.22可得 $id = \varphi * 1$, 两边同时乘上 μ 可得 $id * \mu = \varphi$, 所以 $\varphi^{-1} = id^{-1} * \mu^{-1} = id^{-1} * 1$。
- $\sigma_\alpha * (\sum_{d|n} \mu(d)\mu(\frac{n}{d})d^\alpha) = \epsilon$
  $\sigma_\alpha = id^\alpha * 1$ 可推出 $(\sigma_\alpha)^{-1} = (id^\alpha)^{-1} * \mu$

以上内容参考了 Wikipedia-EN[17]。

4.8.3 莫比乌斯反演

定理 4.27 对于数论函数 f, g, 满足 $g(n) = \sum_{d|n} f(d)$, 则有

$$f(n) = \sum_{d|n} \mu(d)g(\frac{n}{d})$$

莫比乌斯反演可表示为若 $g = f * 1$ 则 $f = \mu * g$。证明: 将 $g = f * 1$ 两边同时乘上 μ 可证。证明源自 Wikipedia-EN[18]。

[17] Dirichlet convolution - Wikipedia
https://en.wikipedia.org/wiki/Dirichlet_convolution
[18] Möbius inversion formula - Wikipedia
https://en.wikipedia.org/wiki/Mobius_inversion

4.8.4 常见技巧

- 对于数论函数 g, f,

$$g(n) = \sum_{n|d} f(d) \Rightarrow f(n) = \sum_{n|d} \mu(d) g(\frac{d}{n})$$

- 若 $n = \prod_{i=1}^{m} {p_i}^{c_i}, g(n) = \sum_{d|n} f(d)$ 且 f 为积性函数, 将 g 看做 $f * 1$ 可知 g 也是积性函数, 则 $g(n) = \prod_{i=1}^{m} \sum_{j=0}^{c_i} f(p_i^j)$。
- 交换内外求和顺序, 尤其是当内部求和项套有不好处理的函数时。
- 枚举倍数、因子、最大公约数等有共性的值并换元。
- 在化简前缀和函数时可能会遇到如下式子:

$$\begin{aligned} ans(n) &= \sum_{i=1}^{n} f(i) \\ &= A(n) + B(n) \sum_{i=1}^{n} \sum_{d|i} f(d) \\ &= A(n) + B(n) \sum_{\frac{i}{d}=1}^{n} \sum_{j=1}^{\left[\frac{n}{\frac{i}{d}}\right]} f(j) \\ &= A(n) + B(n) \sum_{t=1}^{n} \sum_{j=1}^{[\frac{n}{t}]} f(j) \\ &= A(n) + B(n) \sum_{t=1}^{n} ans([\frac{n}{t}]) \end{aligned}$$

  线性筛预处理一部分前缀和(一般预处理规模为 $n^{2/3}$, 最终时间复杂度 $O(n^{2/3})$, 大规模前缀和使用根号分块法递归计算。

  注意这里可以使用存储 Trick 来 Cache 计算结果(多次询问使用 map 或时间戳数组清零, 下面只讨论单次询问的情况)。设预处理了前 k 个前缀和, 其中 $k \geq \sqrt{n}$。那么 $[\frac{n}{t}] > k$ 的值不超过 $\sqrt{n}$ 个, 并且 t 不同对应的 $[\frac{n}{t}]$ 值也不同。所以可以以 t 为下标把计算结果存入另一个数组中。

- 同时除以最大公约数使其互质, 然后套用 φ。
- $[gcd(i,j) = 1] = \sum_{k|gcd(i,j)} \mu(k) = \sum_{k|i,k|j} \mu(k)$
- $gcd(i,j) = \sum_{k|gcd(i,j)} \varphi(k) = \sum_{d=1}^{min(i,j)} \varphi(d)[\frac{i}{d}][\frac{j}{d}]$

  遇到该式时不要枚举 gcd 值将其转换为上一个式子, 从而陷入更复杂的化简。

- $\sum_{i=1}^{n} i = \sum_{i=1}^{n} \sum_{d|i} \varphi(d) = \sum_{d=1}^{n} \varphi(d) \cdot [\frac{n}{d}]$
- $(id \cdot \varphi) * id = id^2$
- 「CQOI2015」选数:

  将 $[L, H]$ 内的 K 的倍数除以 K 得到区间 $[l, r]$, 记 G_i 为 $[l, r]$ 中 i 的倍数个数, 使用前文所述技巧推出答案为 $\sum_{d=1}^{r} \mu(d) G_d^n$。但是 r 仍然是 $1e9$ 级别的, 可以考虑整除分块枚举相同的 G_d 再使用杜教筛求 μ 的前缀和解决。

  注意这里还有个条件 $H - L \leq 1e5$, infinity_edge 的博客[19]中提到存在一个性质:**若所有数字不全相同,则数字的极差不小于它们的最大公约数。**

  接下来只处理数字不全相同的情况, 由于选取的数在 $[l, r]$ 内, 因此只需枚举到 $r - l$, 同时选取方案要扣除数字全相同的情况, 即 $G_d^n - G_d$。这种讨论还有个例外: 数字全为 1 时是可行解, 最后需要特判 l 是否为 1。时间复杂度 $O((r-l) + \sqrt{r-l} \lg N)$。
- 「LibreOJ Round #4」求和:

  $\sum_{d|n} \mu^2(d) \mu(\frac{n}{d})$ 当且仅当 n 为完全平方数时值为 $\mu(\sqrt{n})$, 其余情况为 0。

  证明: 考虑 n 的质因数分解, 若存在质因数幂次 > 2, 则必有一个 μ 值为 0, 该式的值为 0; 若存在质因数幂次为 1, 不考虑其它质因数是否使 μ 为 0, $\mu(\frac{n}{d})$ 项必然抵消。

  综上所述, n 为完全平方数时才有贡献。
- $\mu(ab) = \mu(a)\mu(b)[(a, b) = 1]$

  证明: 当 $(a, b) \neq 1$ 时, ab 有平方因子, 值为 0。否则根据积性函数的性质, $\mu(ab) = \mu(a)\mu(b)$。
- LOJ#6027. 「from CommonAnts」质数计数 I

  考虑构造一个积性函数使得 $f(p) = [p \equiv 1 \pmod 4], f(ab) = f(a)f(b)$。只令 $f(x) = [p \equiv 1 \pmod 4]$ 是错误的, 因为两个在模 4 余 3 的剩余系的元素之积在余 1 剩余系, 这样设计积性函数仅会漏统计这种情况。也就是说要同时统计余 1 和余 3, 但是必须将他们区分。注意积性函数的值域是复数域, 那么可以将积性函数值表示为 $a + bi$ 的形式, 当 $x \equiv 1 \pmod 4$ 时值为 1, 当 $x \equiv 1 \pmod 4$ 时值为 i, 其余情况为 0。再讨论 $f(ab) = f(a)f(b)$ 的性质, 当取 $i^2 = 1$ 时 $f(x)$ 恰好完全积性(不管 i 取什么值都可以构成群, 同样满足积性函数性质)。最后套 min_25 筛解决。事实上还可以将复数扩展为多个基。
- $$\sum_{d=1}^{n} \sum_{k=1}^{\frac{n}{d}} F(k) G(\frac{n}{kd}) \Rightarrow kd = q \Rightarrow \sum_{q=1}^{n} G(q) \sum_{k|q} F(k)$$
- SPOJ DIVCNT2:

  记 $\omega(d)$ 为 d 的质因子个数。

  $$\tau(n^2) = \sum_{d|n} 2^{\omega(d)}$$

[19]「BZOJ3930」「CQOI2015」选数 https://blog.csdn.net/infinity_edge/article/details/78829630

证明:考虑 n^2 有而 n 没有的因子,这些因子必定满足其质因子分解中某个质因子指数超过 n 的对应指数。对于 n 的某个因子 d,其质因子次数加上 n 的对应次数,就可以成为 n^2 的独有因子,方案数为 $2^{\omega(d)}$,且这种计数方法不会重复或遗漏。

$$2^{\omega(n)}=\sum_{d|n}|\mu(d)|=\sum_{d|n}\mu^2(d)$$

证明:n 的所有无平方因子的因子都可以表示为一个长度为 $\omega(n)$ 的 01 向量,表示选/不选这个质因子。枚举所有这种因子等价于枚举所有向量,其集合大小为 $2^{\omega(n)}$。

$$\mu^2(i)=\sum_{j^2|i}\mu(j)$$

证明: 左式的意义是 i 是否含有非平凡平方因子。若 i 不含平方因子, 右式的值为 $\mu(1)=1$。若 i 含有平方因子,记 $i=a^2b,b$ 不含平方因子,那么右式变形为 $\sum_{j|a}\mu(j)$,因为 $a\neq 1$,所以该式的值为 0。

这些方法参考了 Candy? 的博客[20]。

- SDOI2018 旧试题:

$$\begin{aligned}\tau(ij) &= \sum_{x|i}\sum_{y|j}[(x,y)=1]\\ \tau(ijk) &= \sum_{x|i}\sum_{y|j}\sum_{z|k}[(x,y)=1][(x,z)=1][(y,z)=1]\end{aligned}$$

第一个式子证明: 考虑枚举 i 的因子 x, 枚举 j 的因子 y, 这样 xy 可以组成 ij 的因子。但是 xy 可能会有重复, 强制 $z=xy$ 只能被最小的 x 枚举, 那么对于更大的 $x'|z\ x|x'$, 对应的 y' 与 y 相比少了因子 $\frac{x'}{x}$, 注意到 $\frac{j}{y'}$ 多了因子 $\frac{x'}{x}$, 那么 $\frac{j}{y'}$ 与 x' 不互质。而对于最小的 x, 若它与 $\frac{j}{y}$ 不互质, 则令 $x'=\frac{x}{(x,\frac{j}{y})}$ 可以得到更小的方案, 与 x 最小矛盾。那么答案为 $\sum_{x|i}\sum_{y|j}[(x,\frac{j}{y})=1]$, 变形后得到原等式。

- 尽可能早地找到递归结构
- $[(ab,c)=1]=[(a,c)=1][(b,c)=1]$

更多技巧待补充。

4.9 低于线性时间复杂度筛法

积性函数前缀和算法的思维难度:杜教筛 >min_25 筛 >Powerful Number。

4.9.1 杜教筛

杜教筛主要用于计算大数据规模积性函数求和。

[20]SPOJ DIVCNT2 [我也不知道是什么分类了反正是数论] https://www.cnblogs.com/candy99/p/6715013.html

4.9.1.1 约数函数前缀和

求 $\sum_{i=1}^{n}\sigma(i), n \le 10^{12}$。

$$\begin{aligned}\sum_{i=1}^{n}\sigma(i) &= \sum_{i=1}^{n}\sum_{d|i} d \\ &= \sum_{d=1}^{n} d[\frac{n}{d}]\end{aligned}$$

由于 $[\frac{n}{d}]$ 存在许多连续相同的值, 使用整除分块法可做到 $O(\sqrt{n})$。

4.9.1.2 欧拉函数前缀和

求 $\sum_{i=1}^{n}\varphi(i), n \le 10^{11}$。由定理 4.22可得 $\varphi(n) = n - \sum_{d|n,d<n}\varphi(d)$。

$$\begin{aligned}ans(n) &= \sum_{i=1}^{n}\varphi(i) \\ &= \sum_{i=1}^{n}\left(i - \sum_{d|i,d<i}\varphi(d)\right) \\ &= \frac{n(n+1)}{2} - \sum_{i=2}^{n}\sum_{d|i,d<i}\varphi(d) \\ &= \frac{n(n+1)}{2} - \sum_{\frac{i}{d}=2}^{n}\sum_{d=1}^{[\frac{n}{\frac{i}{d}}]}\varphi(d) \\ &= \frac{n(n+1)}{2} - \sum_{t=2}^{n}\sum_{d=1}^{[\frac{n}{t}]}\varphi(d) \\ &= \frac{n(n+1)}{2} - \sum_{t=2}^{n} ans([\frac{n}{t}])\end{aligned}$$

同理使用分块 + 递归询问区间和来计算答案。为了降低复杂度, 应该先线性筛预处理前一部分值。当预处理 $k = n^{\frac{2}{3}}$ 时可以取到复杂度 $T(n) = O(n^{\frac{2}{3}})$。

4.9.1.3 莫比乌斯函数前缀和

求 $\sum_{i=1}^{n}\mu(i), n \le 10^{11}$。由定理 4.21可得 $\mu(n) = [n=1] - \sum_{d|n,d<n}\mu(d)$。

$$\begin{aligned} ans(n) &= \sum_{i=1}^{n}\mu(i) \\ &= \sum_{i=1}^{n}\left([i=1] - \sum_{d|i,d<i}\mu(d)\right) \\ &= 1 - \sum_{i=1}^{n}\sum_{d|i,d<i}\mu(d) \\ &= 1 - \sum_{t=2}^{n} ans([\frac{n}{t}]) \end{aligned}$$

4.9.1.4 其它函数前缀和

主要思路是使用狄利克雷卷积构造出一个简单的前缀和函数, 且用于卷积的另一个函数也容易计算。

令 $A(n) = \sum_{i=1}^{n}\frac{i}{(n,i)}$, 求 $\sum_{i=1}^{n}A(n), n \le 10^9$。

先化简 $A(n)$:

$$\begin{aligned} A(n) &= \sum_{i=1}^{n}\frac{i}{(n,i)} \\ &= \sum_{d|n}\sum_{i=1}^{n}[(n,i)=d]\cdot\frac{i}{d} \\ &= \sum_{d|n}\sum_{\frac{i}{d}=1}^{\frac{n}{d}}[(\frac{n}{d},\frac{i}{d})=1]\cdot\frac{i}{d} \\ &= \frac{1}{2}\left(1 + \sum_{d|n} d\cdot\varphi(d)\right) \end{aligned}$$

那么答案即为 $\frac{1}{2}\left(n + \sum_{t=1}^{n}\sum_{d=1}^{[\frac{n}{t}]} d\cdot\varphi(d)\right)$。

考虑计算 $\sum_{d=1}^{n} d\cdot\varphi(d)$ 的值:

易知 $(id\cdot\varphi) * id = id^2$, 因为

$$\sum_{d|n} d\cdot\varphi(d)\cdot\frac{n}{d} = n\cdot\sum_{d|n}\varphi(d) = n^2$$

所以有

$$\begin{aligned}\frac{n(n+1)(2n+1)}{6} &= \sum_{i=1}^{n}(id\cdot\varphi)*id \\ &= \sum_{t=1}^{n} t\cdot\sum_{d=1}^{[\frac{n}{t}]} d\cdot\varphi(d)\end{aligned}$$

4.9.1.5 总结

欲求积性函数 $f(x)$ 的前缀和, 构造 $h=f*g$, 其中 $h(x)$ 和 $g(x)$ 都是积性函数, 且易求得 $h(x)$ 与 $g(x)$ 的前缀和。

考虑 $h(x)$ 前缀和的表达式(记大写形式为前缀和, 即 $F(n)=\sum_{i=1}^{n} f(i)$):

$$H(n)=\sum_{i=1}^{n}\sum_{d|i} f(\frac{i}{d})g(d)=\sum_{d=1}^{n} g(d)\sum_{i=1}^{\lfloor\frac{n}{d}\rfloor} f(i)=\sum_{d=1}^{n} g(d)F(\lfloor\frac{n}{d}\rfloor)$$

提出 $F(n)$, 即 $F(n)=H(n)-\sum_{d=2}^{n} g(d)F(\lfloor\frac{n}{d}\rfloor)$。

在递归时可以预处理一部分前缀和, 同时使用 HashTable 缓存计算结果。

4.9.1.6 时间复杂度分析

使用整除分块法需要计算前 i 项前缀和, 与前 $\lfloor\frac{n}{i}\rfloor$ 项前缀和, 其中 $i\le\sqrt{n}$。后一部分比前一部分复杂度更高。考虑使用积分近似, 有

$$\int_0^{\sqrt{n}}\sqrt{\frac{n}{x}}\mathrm{d}x=O(n^{3/4})$$

预处理一部分前缀和可以有效降低算法时间复杂度:记预处理前 k 个前缀和, $k\ge\sqrt{n}$。那么时间复杂度为

$$O(k)+\int_0^{\frac{n}{k}}\sqrt{\frac{n}{x}}\mathrm{d}x=O(k)+O(\frac{n}{\sqrt{k}})$$

平衡两边的复杂度, 解得 $k=n^{1/3}$。

以上例题来自 skywalkert 的博客[21], 总结部分参考了国家集训队 2016 论文集任之洲的论文《积性函数求和的几种方法》。

4.9.2 min_25 筛

这里求和的积性函数 F 满足 $F(p)$ 是一个关于 p 的低阶多项式且能够快速求出 $F(p^k)$。据说 min_25 筛踩爆洲阁筛, 那我就不学洲阁筛了。在此附上洲阁筛教程[22]。

[21]浅谈一类积性函数的前缀和 https://blog.csdn.net/skywalkert/article/details/50500009

[22]洲阁筛学习 | ___debug's Home http://debug18.com/posts/calculate-the-sum-of-multiplicative-function/

链接已失效, 不过瞎翻博客后吃到了狗粮, 好羡慕他能和他的女朋友一起上清华。我。。。见后记。祝福他们。

4.9.2.1 预处理

首先考虑求 $\sum_{p\le n} F(p)$。

记 $g(n,j)$ 为满足 x 为 n 以内素数，或者 x 的最小质因子 $>p_j$ 的 $F(x)$ 之和，所求值即为 $g(n,|P|)$。考虑 $g(n,j)$ 如何从 $g(n,j-1)$ 转移。易知最小质因子为 p_j 的合数为 p_j^2，若其 $>n$，则 $g(n,j)$ 与 $g(n,j-1)$ 都只求素数的积性函数值之和，所以 $g(n,j)=g(n,j-1)$。若 $p_j^2\le n$，则转移时会损失掉一些 $F(x)$，这些 x 的最小质因子为 p_j。考虑提出 x 的 p_j，满足 $\frac{x}{p_j}$ 的最小质因子 $\ge p_j$，计算 $\frac{x}{p_j}$ 的积性函数和，发现 $g(\frac{n}{p_j},j-1)$ 包括了它们，又因为 $\frac{n}{p_j}\ge p_j>p_{j-1}$，$g(\frac{n}{p_j},j-1)$ 还有 $\sum_{p<p_j} F(p)$，需要扣除。**由于积性函数 F 的特殊性，把不同次数的项拆开算，单项为完全积性函数，乘上 $F(p_j)$ 即为需要减去的值。**

若拆开后某一项系数不为 1，这一项就不是完全积性。算这一项的贡献时先去除系数，最后整体乘以系数。

综上，有

$$g(n,j)=\begin{cases} g(n,j-1) & p_j^2>n \\ g(n,j-1)-F(p_j)(g(\frac{n}{p_j},j-1)-\sum_{p<p_j}F(p)) & p_j^2\le n \end{cases}$$

预处理素数时只需要筛 $\sqrt{n}$ 内的素数，边界 $g(n,0)$ 是所有数按照素数的计算方式计算的值之和。由于最后只需要 $g(n,|P|)$，非质数的贡献会被筛掉。

实质上 $g(n,j)$ 就是埃氏筛法筛完 p_j 后未被筛的合数以及素数的积性函数值之和。

接下来尝试求出所有的 $g(x,|P|),x=\lfloor\frac{n}{i}\rfloor$。这里有一个存储上的 trick：由于 $\lfloor\frac{n}{i}\rfloor$ 有连续重复项，最多 $2\sqrt{n}$ 个，对于 $x=\lfloor\frac{n}{i}\rfloor>\sqrt{n}$，把它映射到 $\lfloor\frac{n}{x}\rfloor$ 上存储，这样保证了空间复杂度为 $O(\sqrt{n})$。

由于最后只要 $g(x,|P|),g$ 数组只要开 1 维滚动更新。

伪代码如下：

```
int g[2][sqsiz],q[2*sqsiz];
int& getG(int x) {
    if(x<=sqr) return g[0][x];
    return g[1][n/x];
}
void calcG() {
    int m=0,i=1;
    while(i<=n) {
        int val=n/i;
        q[++m]=val;
        getG(val)=sumf(val);
        i=n/val+1;
    }
    for(int i=1;i<=psiz;++i) {
        int cp=p[i],cp2=cp*cp;
        for(int j=1;j<=m && cp2<=q[j];++j) {
            int k=q[j],&val=getG(k);
            val=sub(val,mul(f(cp),getG(k/cp)-sumpf[i-1]));
        }
    }
}
```

计算 G 时始终不考虑 1，在求和时才加入。

4.9.2.2 求和

记 $S(n,j)$ 为 n 以内最小质因子 $\geq p_j$ 的积性函数值和，所求答案即为 $S(n,1)+f(1)$。把 $S(n,j)$ 分为素数和合数求解：

- 对于素数部分，$g(n,|P|)$ 代表了素数积性函数值和，再扣去不满足最小质因子要求的素数，最终贡献为 $g(n,|P|)-\sum_{p<p_j}F(p)$。
- 对于合数部分，枚举其最小质因子 p_k 及其幂次 c，单独贡献为 $F(p_k^c)S(\frac{n}{p_k^c},k+1)+F(p_k^{c+1})$。注意此处的 $F\cdot S$ 直接利用了积性函数的定义，因为 S 部分无 p_k 因子。由于 S 不处理 $j=k$ 的部分，需要另外加上 $F(p_k^{c+1})$。

递归的边界条件为 $n\leq 1\vee n<p_j$，无需记忆化。

时间复杂度为 $O(\frac{n^{\frac{3}{4}}}{\lg n})$，空间复杂度为 $O(\sqrt{n})$。

模板(LOJ#6053. 简单的函数)：

```
#include <cmath>
#include <cstdio>
const int mod = 1000000007, inv2 = 500000004;
int add(int a, int b) {
    a += b;
    return a < mod ? a : a - mod;
}
typedef long long Int64;
int clamp(Int64 x) {
    x %= mod;
    return x >= 0 ? x : x + mod;
}
Int64 n, sqr;
const int size = 100005;
int p[size], sp[size], pcnt = 0;
bool flag[size];
void pre(int n) {
    for(int i = 2; i <= n; ++i) {
        if(!flag[i]) {
            p[++pcnt] = i;
            sp[pcnt] = add(sp[pcnt - 1], i);
        }
        for(int j = 1; j <= pcnt && i * p[j] <= n;
            ++j) {
            flag[i * p[j]] = true;
            if(i % p[j] == 0)
                break;
        }
    }
}
int g[2][size], h[2][size];
Int64 q[2 * size];
int& getG(int (&g)[2][size], Int64 x) {
    if(x <= sqr)
```

```
        return g[0][x];
    return g[1][n / x];
}
Int64 S(Int64 n, int j) {
    if(n <= 1 || p[j] > n)
        return 0;
    int res = clamp(getG(g, n) - sp[j - 1] -
                    (getG(h, n) - (j - 1)) +
                    (j == 1 ? 2 : 0));
    for(int i = j; i <= pcnt; ++i) {
        Int64 cp = p[i], p1 = cp, p2 = p1 * p1;
        if(p2 > n)
            break;
        for(int e = 1; p2 <= n;
            ++e, p1 = p2, p2 *= cp) {
            res = clamp(res +
                        S(n / p1, i + 1) * (cp ^ e) +
                        (cp ^ (e + 1)));
        }
    }
    return res;
}
int main() {
    scanf("%lld", &n);
    sqr = sqrt(n);
    pre(sqr);
    Int64 i = 1;
    int m = 0;
    while(i <= n) {
        Int64 val = n / i;
        q[++m] = val;
        getG(g, val) = ((val + 2) % mod) *
            ((val - 1) % mod) % mod * inv2 % mod;
        getG(h, val) = (val - 1) % mod;
        i = n / val + 1;
    }
    for(int i = 1; i <= pcnt; ++i) {
        Int64 cp = p[i], p2 = cp * cp;
        for(int j = 1; j <= m && p2 <= q[j]; ++j) {
            Int64 k = q[j];
            int &vg = getG(g, k), &vh = getG(h, k);
            vg = clamp(
                vg -
                cp * (getG(g, k / cp) - sp[i - 1]));
            vh = clamp(vh -
                       (getG(h, k / cp) - (i - 1)));
        }
    }
    printf("%d\n", add(S(n, 1), 1));
    return 0;
```

}

上述内容参考了小蒟蒻 yyb[23]和租酥雨[24]的博客。

Min_25 筛似乎也被称作 “通用筛法”、“扩展埃拉托斯特尼筛法”，严格证明参见 zbh2047 的文章[25]。

4.9.3 Powerful Number

定义 Powerful Number 为所有质因子的指数都 ≥ 2 的数，那么每个 Powerful Number 都可以被表示为 a^2b^3 的形式（若指数为奇数则分配一个立方给 b，其余分给 a）。**注意 1 也是 Powerful Number。**

定理 4.28 *n 以内的 Powerful Number 个数为 $O(\sqrt{n})$。*

证明：枚举 a，将 b 的个数累积，可得式子

$$\sum_{i=1}^{\lfloor\sqrt{n}\rfloor}\lfloor\sqrt[3]{\frac{n}{i^2}}\rfloor$$

使用积分近似求出其上界为 $\int_1^{\sqrt{n}+1}\sqrt[3]{\frac{n}{(x-1)^2}}\mathrm{d}x=O(\sqrt{n})$。根据 Wikipedia-EN[26]的描述，其上界常数为 $\frac{\zeta(\frac{3}{2})}{\zeta(3)}\approx 2.173$。

对于某个复杂的积性函数 $f(x)$，若 $f(p^e)$ 易于计算且存在一个简单（易于求 $g(p^e)$ 与前缀和）的积性函数 $g(x)$，满足对于所有素数 p，有 $f(p)=g(p)$，称函数 g 拟合了函数 f。

设 $h=f/g$，这里的除法是狄利克雷除法，等价于 $h=f*g^{-1}$。由于狄利克雷逆 g^{-1} 是积性函数，狄利克雷卷积 h 也是积性函数。那么对于所有素数 p，有 $f(p)=h(1)g(p)+h(p)g(1)$，由于 $h(1)=1, f(p)=g(p)$，可得 $h(p)=0$。由于 $h(x)$ 是积性函数，$h(x)$ 可能非 0 当且仅当 x 是 Powerful Number。

现在要求 $Ans=\sum_{i=1}^{n}f(n)$，由于 $f=h*g$，有 $Ans=\sum_{ab\leq n}h(a)g(b)$。由上文的推导可知 $h(x)$ 仅在 Powerful Number 处有贡献，且 Powerful Number 的个数是 $O(\sqrt{n})$ 的，可以 $O(\sqrt{n})$DFS 暴力枚举质因子组合得到 a。记 n 以内的 Powerful Number 组成的集合为 S，原式变为 $Ans=\sum_{a\in S}h(a)\sum_{b=1}^{\lfloor\frac{n}{a}\rfloor}g(b)$。易求 $g(x)$ 的前缀和，问题在于如何快速推得 $h(a)$ 的值。

由于 $h(x)$ 是积性函数且 x 是 Powerful Number，在 DFS 时仅需计算 $h(p^e), e>1$ 的值。使用 $f(p^e)$ 展开式，快速求得 $f(p^e)$ 与 $g(p^e)$，再根据历史信息 $h(p^{e'}), e'<e$，可以快速得到 $h(p^e)$。如果 $g(x)$ 是完全积性函数，可以对 $f(p^e)$ 展开式平移得到 $f(p^{e+1})$ 的展开式。由于 e 很小，$h(p^e)$ 的求值不是瓶颈。预处理可以节省 DFS 时的重复计算。

十分有效的优化：DFS 递归时会遇到大量的 0 次项，这些不必要的递归会导致实际运行缓慢。可以 DFS 钦定一些质因子必选，使得每层 DFS 都对最终的 a 有贡献，杜绝爆栈。

具体实现参考 Project Euler 484：

---

[23]min_25 筛 https://www.cnblogs.com/cjyyb/p/9185093.html

[24]Min_25 筛 https://www.cnblogs.com/zhoushuyu/p/9187319.html

[25]关于一种积性函数前缀和的通用筛法的时间复杂度证明
https://www.cnblogs.com/zbh2047/p/8552551.html
https://zhuanlan.zhihu.com/p/33544708

[26]Powerful number https://en.wikipedia.org/wiki/Powerful_number

```
#include <cmath>
#include <iostream>
typedef __int128 Int;
typedef long long Int64;
#define asInt static_cast<Int>
const int size = 8e7 + 5;
int p[size], pcnt = 0;
bool flag[size];
void pre(int n) {
    std::cout << "pre " << n << std::endl;
    for(int i = 2; i <= n; ++i) {
        if(!flag[i])
            p[++pcnt] = i;
        for(int j = 1; j <= pcnt &&
            static_cast<Int64>(i) * p[j] <= n;
            ++j) {
            flag[i * p[j]] = true;
            if(i % p[j] == 0)
                break;
        }
    }
    std::cout << "prime " << pcnt << std::endl;
}
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
Int ans = 0;
Int64 n, cnt = 0;
void DFS(int pi, Int a, Int h) {
    Int cp = p[pi], cur = cp, sh = 1;
    a *= cp * cp;
    for(int e = 2; a <= n; a *= cp, cur *= cp, ++e) {
        Int nsh = cur * gcd(cp, e);
        Int ch = nsh - sh;
        sh = nsh;
        Int nh = h * ch;
        ans += nh * (n / a);
        if(((++cnt) & 0xfffff) == 0)
            std::cout << cnt << std::endl;
        for(int i = pi + 1; i <= pcnt; ++i) {
            Int np = p[i];
            if(a * np * np > n)
                break;
            DFS(i, a, nh);
        }
    }
}
void print(Int x) {
    if(x >= 10)
        print(x / 10);
```

```
    putchar('0' + x % 10);
}
int main() {
    std::cin >> n;
    pre(sqrt(n));
    ans = n;
    for(int i = 1; i <= pcnt; ++i)
        DFS(i, 1, 1);
    print(ans - 1);
    putchar('\n');
    std::cout << "Powerful Number Count " << cnt
              << std::endl;
    return 0;
}
```

记得要计算 $a = 1$ 时的贡献!!!

上述内容参考了 fjzzq2002 的博客[27]。

Min_25 使用 Powerful Number 得到新的做法, 参见 Sum of Multiplicative Function on Powerful Numbers [28]。

4.9.4 素数 k 次幂前缀和

可以使用 Min_25 筛的前半部分在 $O(\frac{n^{3/4}}{\lg n})$ 内解决, 但这不是最优的。

一般使用 Meissel Lehmer 方法在 $O\left(\left(\frac{n}{\lg n}\right)^{2/3}\right)$ 内解决, 留坑待补。

4.9.5 约数个数函数前缀和

使用整除分块可以在 $O(\sqrt{n})$ 内解决, 但还有更优算法。

考虑变化后的式子 $S(n) = \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor$, 可以将其理解为在 $[1, n]$ 内在双曲线 $xy = n$ 与 x 轴内的整点数。使用 Stern-Brocot Tree 可以在 $O(n^{1/3} \lg n)$ 内解决, 留坑待补。

4.10 离散对数问题

离散对数问题是求出方程 $a^x \equiv b \pmod{p}$ 的可行解/最小非负整数解。

4.10.1 BSGS

4.10.1.1 BSGS

普通 BSGS 仅考虑 P 为素数的情况。

以下为求解最小非负整数解的方法:

首先根据定理 4.9可知 x 的最小非负整数解小于 $\varphi(P) = P - 1$。将 x 表示为 $\sqrt{P}$ 进制数, 分别用 $O(\sqrt{P})$ 的复杂度枚举值, 一半存入 HashTable, 另一半查询是否有匹配值。注意参数的枚举顺序。

1. 若 a 为 P 的倍数, 则特判 b 是否为 0, 算法结束;

[27] 利用 powerful number 求积性函数前缀和 https://www.cnblogs.com/zzqsblog/p/9904271.html

[28] https://min-25.hatenablog.com/entry/2018/11/11/172228

2. 令 $m = \lceil \sqrt{P} \rceil, x = im - j$, 移项得 $a^{im} \equiv ba^j \pmod P$;
3. 枚举 ba^j 的值, 按 j 从小到大**覆盖**存入 HashTable;
4. 枚举 $(a^m)^i$ 的值, 按 i 从小到大在 HashTable 中查询, 存在则返回 $im - j$;
5. 返回无解。

今后将采用 3.11节的双重散列 + 开放寻址法,严禁使用 std::unordered_map。

4.10.1.2 ExBSGS

ExBSGS 可解决 a, P 不互质的问题。主要思路是将原方程化为普通 BSGS 可解决的方程。

记化简后方程为 $Aa^{x-B} \equiv b \pmod P$, 化简步骤如下:

1. 将 A, B 初始化为 $1, 0$;
2. 令 $d = (a, P)$,
   - 若 $d \mid b$, 则提出一个因子 d, 即 $A* = a/d, b/ = d, P/ = d, ++B$;
   - 若 $d \nmid b$, 则特判 b 是否为 $A, b = A$ 则 $x = B, b \neq A$ 则无解;
3. 重复第 2 步直至 $d = 1$。

令 $x = im - j + B$ 转化为普通 BSGS, **注意在 BSGS 前要暴力检查 $x \in [0, B)$ 是否可行**。

代码如下:

```
#include <cmath>
#include <cstdio>
#include <unordered_map>
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
Int64 powm(Int64 a, int k, Int64 p) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % p;
        k >>= 1, a = a * a % p;
    }
    return res;
}
typedef std::unordered_map<int, int> Hash;
typedef Hash::iterator IterT;
int foo(int a, int p, int b) {
    a %= p, b %= p;
    int A = 1, B = 0;
    while(true) {
        int d = gcd(a, p);
        if(d == 1)
```

```
            break;
        if(b % d) {
            if(a == b)
                return B;
            return -1;
        } else
            ++B, b /= d, p /= d,
                A = asInt64(A) * (a / d) % p;
    }
    Int64 cur = 1;
    for(int i = 0; i < B; ++i) {
        if(cur == b)
            return i;
        cur = cur * a % p;
    }
    int m = sqrt(p) + 1;
    cur = b;
    Hash hash;
    for(int i = 1; i <= m; ++i) {
        cur = cur * a % p;
        hash[cur] = i;
    }
    Int64 base = powm(a, m, p);
    cur = A;
    for(int i = 1; i <= m; ++i) {
        cur = cur * base % p;
        IterT it = hash.find(cur);
        if(it != hash.end())
            return i * m - it->second + B;
    }
    return -1;
}
int main() {
    int a, p, b;
    while(true) {
        scanf("%d%d%d", &a, &p, &b);
        if(a | p | b) {
            int res = foo(a, p, b);
            if(res >= 0)
                printf("%d\n", res);
            else
                puts("No Solution");
        } else
            break;
    }
    return 0;
}
```

以上内容参考了 ZigZagK 的博客[29]。

[29]BSGS 及扩展 BSGS

4.10.2 单有原根模数多询问离散对数问题

对于这类问题,每次都计算一次 HashTable 十分浪费。考虑求出原根 g,令 $a = g^A, b = g^B$,原式转化为 $Ax \equiv B \pmod{\varphi(n)}$,使用 exgcd 快速解决。由于求 a, b 的对数时的底数固定为 g,只要预处理 g 的 BSGS 哈希表(哈希表的大小不必开根号,可以根据模数与询问规模决定)。

若模数 p 非常大,参见 LOJ6542https://loj.ac/problem/6542,留坑待补。

4.10.3 Pollard rho 算法

该算法的主要思路在于找到两组指数 $(y_1, y_2), (z_1, z_2)$,使得 $a^{y_1}b^{y_2} \equiv a^{z_1}b^{z_2}$。然后将其转化为求解模线性方程组 $(z_2 - y_2)x \equiv y_1 - z_1 \pmod{N}$,这里的 N 是循环群的阶。exgcd 轻松解决,不过需要注意的是要检查每个周期的解的合法性(最小非负整数解不一定是可行解)。

若模数有原根且不知道循环群的阶,可以求出 n 的原根 g,那么阶 $N = \varphi(n)$。

找指数的过程类似于因子分解的 PollardRho 算法,即使用随机函数来生成两组指数,同时使用 Folyd 判圈法终止算法。

具体做法是将当前乘积 v 的集合划分到 3 个集合,不同集合的行为不同,分别为 a 的指数 $+1$, b 的指数 $+1$, a, b 的指数翻倍。

该算法的期望复杂度是 $O(\sqrt{n})$,代码量小,但是慢于 BSGS。不过 $O(1)$ 的空间使得它在求解 n 较大的情况时较 BSGS 有绝对优势。

代码如下:

```
#include <algorithm>
#include <cstdio>
using Int64 = long long;
#define asInt64 static_cast<Int64>
inline Int64 addm(Int64 a, Int64 b, Int64 mod) {
    a += b;
    return a < mod ? a : a - mod;
}
inline Int64 mulm(Int64 a, Int64 b, Int64 mod) {
    return (a * b -
            asInt64(static_cast<long double>(a) / mod *
                    b) *
                mod +
            mod) %
        mod;
}
Int64 powm(Int64 a, Int64 k, Int64 mod) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = mulm(res, a, mod);
        k >>= 1, a = mulm(a, a, mod);
    }
    return res;
}
void next(Int64& v, Int64& x, Int64& y, Int64 p,
```

https://blog.csdn.net/zzkksunboy/article/details/73162229

```
            Int64 phi, Int64 a, Int64 b) {
    switch(v % 3) {
        case 0:
            v = mulm(v, v, p), x = addm(x, x, phi),
            y = addm(y, y, phi);
            break;
        case 1:
            v = mulm(v, a, p), x = addm(x, 1, phi);
            break;
        case 2:
            v = mulm(v, b, p), y = addm(y, 1, phi);
            break;
    }
}
void exgcd(Int64 a, Int64 b, Int64& d, Int64& x,
           Int64& y) {
    if(b) {
        exgcd(b, a % b, d, y, x);
        y -= a / b * x;
    } else
        x = 1, y = 0, d = a;
}
std::pair<Int64, Int64> solve(Int64 a, Int64 b,
                              Int64 p) {
    Int64 d, x, y;
    exgcd(a, p, d, x, y);
    if(b % d)
        return std::make_pair(-1LL, 0LL);
    Int64 modv = p / d;
    if(modv < 0)
        modv = -modv;
    x %= modv;
    if(x < 0)
        x += modv;
    Int64 c = (b / d) % modv;
    if(c < 0)
        c += modv;
    return std::make_pair(mulm(x, c, modv), modv);
}
Int64 PollardRho(Int64 a, Int64 b, Int64 p) {
    a %= p, b %= p;
    Int64 phi = p - 1;
    Int64 v1 = 1, x1 = 0, y1 = 0;
    Int64 v2 = 1, x2 = 0, y2 = 0;
    for(Int64 i = 0; i < phi; ++i) {
        next(v1, x1, y1, p, phi, a, b);
        next(v2, x2, y2, p, phi, a, b);
        next(v2, x2, y2, p, phi, a, b);
        if(v1 == v2) {
            Int64 ka = x1 - x2;
```

```
            if(ka < 0)
                ka += phi;
            Int64 kb = y2 - y1;
            if(kb < 0)
                kb += phi;
            auto pair = solve(kb, ka, phi);
            if(pair.first == -1)
                return -1;
            for(Int64 x = pair.first; x < phi;
                x += pair.second)
                if(powm(a, x, p) == b)
                    return x;
            return -1;
        }
    }
    return -1;
}
Int64 calcG(Int64 p) {
    Int64 fac[70], x = p - 1;
    int fcnt = 0;
    for(Int64 i = 2; i <= x; ++i)
        if(x % i == 0) {
            do
                x /= i;
            while(x % i == 0);
            fac[++fcnt] = i;
        }
    for(Int64 i = 2; i < p; ++i) {
        bool flag = true;
        for(int j = 1; j <= fcnt; ++j)
            if(powm(i, p / fac[j], p) == 1) {
                flag = false;
                break;
            }
        if(flag)
            return i;
    }
    return -1;
}
Int64 solveDL(Int64 a, Int64 b, Int64 p, Int64 g) {
    Int64 phi = p - 1;
    Int64 lga = PollardRho(g, a, p);
    if(lga == -1)
        return -1;
    Int64 lgb = PollardRho(g, b, p);
    if(lgb == -1)
        return -1;
    return solve(lga, lgb, phi).first;
}
int main() {
```

```
    int t;
    Int64 p;
    scanf("%d%lld", &t, &p);
    Int64 g = calcG(p);
130 for(int i = 0; i < t; ++i) {
        Int64 a, b;
        scanf("%lld%lld", &a, &b);
        printf("%lld\n", solveDL(a, b, p, g));
    }
    return 0;
}
```

上述内容参考了 Wikipedia-EN[30]。

4.10.4 Pohlig-Hellman 算法

该算法的思路基于群论，需要知道循环群的阶。如果循环群的阶是一个 smooth number，即可以分解为小素数幂之积，就能够将其分解为小规模的子问题。如果模数有原根，就可以沿用上面的做法。记循环群的阶 $n = \prod p_i^{e_i}$，该算法可以将复杂度降到 $O(\sum e_i(\lg n + \sqrt{p_i}))$，空间复杂度 $O(\sqrt{p_{max}})$。

4.10.4.1 阶为素数幂的情况

记素数幂为 p^e，算法步骤如下：

1. 初始化 $x_0 = 0$。
2. 计算 $G = g^{p^{e-1}}$，根据定理 5.5，生成元 G 对应的循环群的阶为 p。
3. 对于 $k = 0, 1, \cdots, e-1$，计算 $b_k = (g^{-x_k}b)^{p^{e-1-k}}$，BSGS 求解 $G^{d_k} \equiv b_k \pmod n$（注意模数是 n，阶为 p，时空复杂度 $O(\sqrt{p})$），最后累加 $x_{k+1} = x_k + p^k d_k$。
4. 返回 x_e。

算法正确性留坑待补。

4.10.4.2 一般情况

首先将阶 n 因式分解，求出模数的原根 g。然后逐个计算 $g_i = g^{n/p_i^{e_i}}$，构造出阶为 p^e 形式的群，对应地 $b_i = b^{n/p_i^{e_i}}$，使用上述方法求出 $g^{x_i} \equiv b_i \pmod{p_i^{e_i}}$ 的解，最后使用 CRT 合并答案。

代码如下（针对单模数多询问重新平衡，以空间换时间）：

```
0 #include <cmath>
  #include <cstdio>
  #include <unordered_map>
  #include <vector>
  using Int64 = long long;
  using HashTable = std::unordered_map<Int64, int>;
  using IterT = HashTable::const_iterator;
```

[30] Pollard's rho algorithm for logarithms https://en.wikipedia.org/wiki/Pollard's_rho_algorithm_for_logarithms

```
#define asInt64 static_cast<Int64>
inline Int64 addm(Int64 a, Int64 b, Int64 mod) {
    a += b;
    return a < mod ? a : a - mod;
}
#ifdef ONLINE_JUDGE
using Int128 = __int128;
inline Int64 mulm(Int128 a, Int128 b, Int128 mod) {
    return a * b % mod;
}
#else
inline Int64 mulm(Int64 a, Int64 b, Int64 mod) {
    return (a * b -
            asInt64(static_cast<long double>(a) / mod *
                    b) *
                mod +
            mod) %
        mod;
}
#endif
Int64 powm(Int64 a, Int64 k, Int64 mod) {
    a %= mod;
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = mulm(res, a, mod);
        k >>= 1, a = mulm(a, a, mod);
    }
    return res;
}
struct Prime {
    Int64 p, pc, bsiz, maxb, k, g, G;
    int c;
    HashTable map;
    Prime(Int64 p, int c, Int64 pc)
        : p(p), pc(pc), c(c) {}
};
auto fac(Int64 p) {
    std::vector<Prime> pfac;
    for(Int64 i = 2; i * i <= p; ++i)
        if(p % i == 0) {
            int pcnt = 0;
            Int64 pc = 1;
            do
                ++pcnt, pc *= i, p /= i;
            while(p % i == 0);
            pfac.push_back(Prime(i, pcnt, pc));
        }
    if(p != 1)
        pfac.push_back(Prime(p, 1, p));
```

```
    return pfac;
}
Int64 calcG(Int64 p, const std::vector<Prime>& fac) {
    for(Int64 i = 2; i < p; ++i) {
        bool flag = true;
        for(int j = 0; j < fac.size(); ++j)
            if(powm(i, p / fac[j].p, p) == 1) {
                flag = false;
                break;
            }
        if(flag)
            return i;
    }
    return -1;
}
void exgcd(Int64 a, Int64 b, Int64& d, Int64& x,
           Int64& y) {
    if(b) {
        exgcd(b, a % b, d, y, x);
        y -= a / b * x;
    } else
        x = 1, y = 0, d = a;
}
Int64 solve(Int64 a, Int64 b, Int64 p) {
    Int64 d, x, y;
    exgcd(a, p, d, x, y);
    if(b % d)
        return -1;
    Int64 c = b / d, modv = p / d;
    if(modv < 0)
        modv = -modv;
    x %= modv;
    if(x < 0)
        x += modv;
    c %= modv;
    if(c < 0)
        c += modv;
    return mulm(x, c, modv);
}
Int64 lookUp(Int64 x, Int64 g, Int64 p,
             const HashTable& map, Int64 bsiz) {
    for(Int64 i = 1, cur = mulm(x, g, p); i <= bsiz;
        ++i, cur = mulm(cur, g, p)) {
        IterT it = map.find(cur);
        if(it != map.cend())
            return it->second * bsiz - i;
    }
    return -1;
}
Int64 log(const Prime& fac, Int64 phi, Int64 mod,
```

```
          Int64 b) {
    Int64 xk = 0, cpc = 1;
    for(int i = 0; i < fac.c; ++i, cpc *= fac.p) {
        Int64 k = -xk % phi;
        if(k < 0)
            k += phi;
        Int64 cb =
            powm(mulm(powm(fac.g, k, mod), b, mod),
                 powm(fac.p, fac.c - 1 - i, phi), mod);
        xk += cpc *
            lookUp(cb, fac.G, mod, fac.map, fac.bsiz);
    }
    return xk;
}
Int64 log(const std::vector<Prime>& fac, Int64 phi,
          Int64 mod, Int64 b) {
    Int64 res = 0;
    for(auto& p : fac) {
        res =
            (res + mulm(log(p, phi, mod,
                            powm(b, phi / p.pc, mod)),
                        p.k, phi)) %
            phi;
    }
    return res;
}
void PohligHellman(int t, Int64 p) {
    Int64 phi = p - 1;
    auto pf = fac(phi);
    Int64 g = calcG(p, pf);
    for(size_t i = 0; i < pf.size(); ++i) {
        Prime& cp = pf[i];
        cp.bsiz = sqrt(cp.p / t) + 1;
        cp.maxb = cp.p / cp.bsiz + 1;
        cp.g = powm(g, phi / cp.pc, p),
        cp.G = powm(cp.g, cp.pc / cp.p, p);
        Int64 base = powm(cp.G, cp.bsiz, p);
        for(Int64 cur = base, i = 1; i <= cp.maxb;
            ++i, cur = mulm(cur, base, p)) {
            if(!cp.map.count(cur))
                cp.map[cur] = i;
        }
        cp.k = mulm(phi / cp.pc,
                    powm(phi / cp.pc,
                         cp.pc / cp.p * (cp.p - 1) - 1,
                         cp.pc),
                    phi);
    }
    for(int i = 0; i < t; ++i) {
        Int64 a, b;
```

```
        scanf("%lld%lld", &a, &b);
        Int64 lga = log(pf, phi, p, a),
              lgb = log(pf, phi, p, b);
        printf("%lld\n", solve(lga, lgb, phi));
    }
}
int main() {
    int t;
    Int64 p;
    scanf("%d%lld", &t, &p);
    PohligHellman(t, p);
    return 0;
}
```

上述内容参考了 Wikipedia-EN[31]。

4.10.5 对 1 求离散对数

沿用判断原根的思路, 可行解 x 一定是 $\varphi(n)$ 的某个因子, 最坏复杂度 $O(\sqrt{n}\lg n)$, 一般情况下比 BSGS 更加简洁高效。

该方法源自 Dance Of Faith 的博客[32]。

4.11 原根

4.11.1 基本定义与定理

4.11.1.1 数论阶

设 $n>1,(a,n)=1$, 记 $\delta_n(a)$ 为使得 $a^r\equiv 1 \pmod n$ 成立的最小正整数 r, 称其为 a 模 n 的阶。

定理 4.29 设 $n>1,(a,n)=1$, 若 $a^x\equiv 1 \pmod n$, 则有 $\delta_n(a)\mid x$。

4.11.1.2 原根

若 $\delta_n(a)=\varphi(n)$, 则称 a 为模 n 的一个原根。

若 a 为模 n 的原根, 根据定理 4.9, 对于 $0\le i<\varphi(n)$, $a^i \bmod n$ 两两不同。

定理 4.30 如果模 n 有原根, 则它一共有 $\varphi(\varphi(n))$ 个原根。

定理 4.31 $n=2,4,p^i,2p^i \Leftrightarrow$ 模 n 有原根, 其中 p 为奇素数。

4.11.2 求模 n 的原根

对 $\varphi(n)$ 进行质因数分解, 对于 $\varphi(n)=\prod_{i=1}^{m} p_i^{c_i}$, 若恒有 $g^{\frac{\varphi(n)}{p_i}}\not\equiv 1 \pmod n$, 则 g 为模 n 的原根。

以上内容参考了 mosquito_zm 的博客[33]。

[31] Pohlig–Hellman algorithm
https://en.wikipedia.org/wiki/Pohlig-Hellman_algorithm

[32] 【科技】 原根的快速判断方法以及对 1 求离散对数的另一种想法https://www.cnblogs.com/Dance-Of-Faith/p/9905786.html

[33] 原根 https://blog.csdn.net/mosquito_zm/article/details/77227570

4.11.3 原根的应用

- 在 NTT 中用于推算主单位根。
- 将乘积恒定转换为幂次和恒定后 NTT。

例题 [SDOI2015] 序列统计[34]

将 x 映射为 g^i, 使用生成函数推导出多项式幂的形式, 最后对答案进行逆映射。

luogu P3321

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
const int size = 17005, mod = 1004535809;
Int64 powm(Int64 a, int k, int mod) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
int add(int a, int b) {
    a += b;
    return a >= mod ? a - mod : a;
}
int sub(int a, int b) {
    a -= b;
    return a < 0 ? a + mod : a;
}
int root[size], invR[size], tot;
void init(int n) {
    const int g = 3;
```

[34]【P3321】[SDOI2015] 序列统计 - 洛谷 https://www.luogu.org/problemnew/show/P3321

```
    tot = n;
    Int64 base = powm(g, (mod − 1) / n, mod);
    Int64 invBase = powm(base, mod − 2, mod);
    root[0] = invR[0] = 1;
    for(int i = 1; i < n; ++i)
        root[i] = root[i − 1] * base % mod;
    for(int i = 1; i < n; ++i)
        invR[i] = invR[i − 1] * invBase % mod;
}
void NTT(int n, int* A, int* w) {
    for(int i = 0, j = 0; i < n; ++i) {
        if(i > j)
            std::swap(A[i], A[j]);
        for(int l = n >> 1; (j ^= l) < l; l >>= 1)
            ;
    }
    for(int i = 2; i <= n; i <<= 1) {
        int m = i >> 1, fac = tot / i;
        for(int j = 0; j < n; j += i) {
            for(int k = 0; k < m; ++k) {
                int t = asInt64(A[j + k + m]) *
                    w[fac * k] % mod;
                A[j + k + m] = sub(A[j + k], t);
                A[j + k] = add(A[j + k], t);
            }
        }
    }
}
int calcG(int p) {
    int fac[15];
    int x = p − 1, fcnt = 0;
    for(int i = 2; i <= x; ++i)
        if(x % i == 0) {
            do
                x /= i;
            while(x % i == 0);
            fac[++fcnt] = i;
        }
    for(int i = 2; i < p; ++i) {
        bool flag = true;
        for(int j = 1; j <= fcnt; ++j)
            if(powm(i, p / fac[j], p) == 1) {
                flag = false;
                break;
            }
        if(flag)
            return i;
    }
    return −1;
}
```

```
int map[10000];
void initMap(int m, int g) {
    int cur = 1, end = m - 2;
    for(int i = 0; i <= end; ++i) {
        map[cur] = i;
        cur = cur * g % m;
    }
}
int A[size] = {}, B[size] = {}, C[size];
void powPoly(int n, int m) {
    int d = m - 1;
    int k = 1;
    while(k < (d << 1))
        k <<= 1;
    init(k);
    Int64 inv = powm(k, mod - 2, mod);
    B[0] = 1;
    while(n) {
        if(n & 1) {
            // B=A*B
            memcpy(C, A, sizeof(int) * k);
            NTT(k, C, root);
            NTT(k, B, root);
            for(int i = 0; i < k; ++i)
                B[i] = asInt64(C[i]) * B[i] % mod;
            NTT(k, B, invR);
            for(int i = 0; i < d; ++i)
                B[i] = add(B[i], B[i + d]) * inv % mod;
            memset(B + d, 0, sizeof(int) * (k - d));
        }
        n >>= 1;
        // A=A*A
        NTT(k, A, root);
        for(int i = 0; i < k; ++i)
            A[i] = asInt64(A[i]) * A[i] % mod;
        NTT(k, A, invR);
        for(int i = 0; i < d; ++i)
            A[i] = add(A[i], A[i + d]) * inv % mod;
        memset(A + d, 0, sizeof(int) * (k - d));
    }
}
int main() {
    int n = read();
    int m = read();
    int g = calcG(m);
    initMap(m, g);
    int x = map[read()];
    int s = read();
    for(int i = 0; i < s; ++i) {
        int x = read();
```

```
            if(x)
                ++A[map[x]];
        }
140     powPoly(n, m);
        printf("%d\n", B[x]);
        return 0;
    }
```

4.12 二次剩余及其扩展

4.12.1 勒让德符号

定义勒让德符号:

$$\left(\frac{a}{p}\right)=\begin{cases}0 & a\equiv 0 \pmod p\\ 1 & \exists x, x^2\equiv a \pmod p\\ -1 & \nexists x, x^2\equiv a \pmod p\end{cases}$$

这里的 p 是奇素数。
勒让德符号是完全积性函数, 即

$$\left(\frac{ab}{p}\right)=\left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$$

4.12.1.1 与斐波那契数列的关系

定理 4.32 若 p 为素数, 则

$$F_{p-\left(\frac{p}{5}\right)}\equiv 0 \pmod p$$

$$F_p\equiv\left(\frac{p}{5}\right) \pmod p$$

该定理用于求超大斐波那契数取模。

4.12.1.2 二次互反律

定理 4.33 若 p,q 为不同的奇素数, 则

$$\left(\frac{p}{q}\right)\left(\frac{q}{p}\right)=(-1)^{\frac{(p-1)(q-1)}{4}}$$

此外有两个补充结论:

定理 4.34

$$\left(\frac{-1}{p}\right)=(-1)^{\frac{p-1}{2}}=\begin{cases}1 & if\ p\equiv 1 \pmod 4\\ -1 & if\ p\equiv 3 \pmod 4\end{cases}$$

定理 4.35

$$\left(\frac{2}{p}\right)=(-1)^{\frac{p^2-1}{8}}=\begin{cases}1 & if\ p\equiv 1,7 \pmod 4\\ -1 & if\ p\equiv 3,5 \pmod 4\end{cases}$$

4.12.2 二次剩余

求解二次剩余即求解下列同余方程:

$$x^2 \equiv a \pmod{p}$$

4.12.2.1 欧拉判别准则

定理 4.36 (Euler's Criterion) 若 p 为奇素数且 $p \nmid a$,则

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p}$$

4.12.2.2 模奇素数

这里使用 ACdreamer 介绍的 Cipolla 随机化算法。

定理 4.37 设 b 满足 $\omega = b^2 - a$ 不是模 p 的二次剩余,则 $x \equiv (b + \sqrt{\omega})^{\frac{p+1}{2}} \pmod{p}$ 是方程 $x^2 \equiv a \pmod{p}$ 的解。

证明:

由推论 6.6可得:

$$(b + \sqrt{\omega})^p \equiv b + \omega^{\frac{p-1}{2}}\sqrt{\omega} \equiv b - \sqrt{\omega} \pmod{p}$$

那么有

$$\begin{aligned}(b + \sqrt{\omega})^{p+1} &\equiv (b + \sqrt{\omega})^p (b + \sqrt{\omega}) \\ &\equiv (b - \sqrt{\omega})(b + \sqrt{\omega}) \\ &= b^2 - \omega \\ &= a \pmod{p}\end{aligned}$$

b 可以随机选取,因为有一半左右的 ω 不是模 p 的二次剩余。然后把 $\sqrt{\omega}$ 当做虚数单位做复数快速幂,实数部分就是答案。该结果的相反数也是一个解。

4.12.2.3 模奇素数幂

即 p 为奇素数且 $(a, p) = 1$,求解下列方程:

$$x^2 \equiv a \pmod{p^n}$$

若 $(a, p) \neq 1$,令 $a = A \cdot p^k$。这里仅考虑 $k < n$ 的非平凡情况。

若 $2 \nmid k$ 则 a 不是模 p^n 的二次剩余,假设 a 是模 p^n 的二次剩余,则存在 $r, s \nmid p$ 满足 $(p^r \cdot s)^2 \equiv A \cdot p^k \pmod{p^n}$,那么有 $p^k \equiv p^{2r} \pmod{p^n}$,该式与假设矛盾。

若 $2 \mid k$ 则将原方程化为 $x_0^2 \equiv A \pmod{p^{n-k}}$,这里满足开头的假设 $(a, p) = 1$,用下面的方法求解 x_0,答案即为 $x_0 \cdot p^{\frac{k}{2}}$。

首先求出 $x^2 \equiv a \pmod{p}$ 的解 r,有 $r^2 - a = kp$。进一步推得 $(r^2 - a)^n \equiv 0 \pmod{p^n}$,将 $r^2 - a$ 分解为 $(r + \sqrt{a})(r - \sqrt{a})$,两边快速幂得 $(r \pm \sqrt{a})^n = u \pm v\sqrt{a}$ (这两个因式是共轭复数, 所以结果也共轭)。由此可得 $(r^2 - a)^n = u^2 - v^2 a \equiv 0 \pmod{p}$,因此最终答案为 $\frac{u}{v}$。这里的 u, v 均与 p 互质,可以扩欧或求出 $\varphi(p^n)$ 后快速幂。

4.12.2.4 模 2 的幂

即求解下列方程:

$$x^2 \equiv a \pmod{2^n}$$

类似地, 仅讨论 $(a, 2) = 1$ 的情况。

- 若 $n = 1$, 当且仅当 $a = 1$ 时有解 1;
- 若 $n = 2$, 当且仅当 $a = 1$ 时有解 ∓ 1;
- 若 $n > 2$, 当且仅当 $a \equiv 1 \pmod 8$ 时有四个解。当 $n = 3$ 时解为 $\mp 1, \mp 5$, 这里只保留一个解 $x = 1$ 用来递推。因为若存在 x_0 满足该方程, 则 $2^n - x_0$, $2^{n-1} - x_0$, $2^{n-1} + x_0$ 也是该方程的解。用递推式 $x_k = \frac{A - x_{k-1}^2}{2} \bmod 2^k$ 求出解。

具体证明留坑待补。

4.12.2.5 一般解法

将模数分解为素数幂之积后 CRT 求解, 注意有多个解。

上述内容参考了 Miskcoo[35]、ACdreamer[36]和 Quack_quack[37] 的博客与百度百科(勒让德符号, 二次互反律与欧拉判别准则)。

4.12.3 三次剩余

首先特判 $a = 0$ 和 $p \le 3$ 的情况。其它处理方法同上文, 下面仅讨论 p 为素数的情况。

若 $p \equiv 2 \pmod 3$ 则方程有唯一解 $x \equiv a^{\frac{2p-1}{3}} \pmod p$。否则当 $\left(\frac{-3}{p}\right) = 1$ 时才有解, 记 $\epsilon = \frac{\sqrt{-3}-1}{2}$ 为三次单位根, 有 $\epsilon^3 \equiv 1 \pmod p$。

类似于二次剩余, 有三分之一左右的数是三次剩余, 且若找到一个解 x, $x\epsilon, x\epsilon^2$ 也是该方程的解。

类比 Cipolla 随机化算法, 随机出一个值 b 满足 $b^3 - a$ 是三次非剩余 (当且仅当 $a^{\frac{p-1}{3}} \equiv 1 \pmod p$ 时为三次剩余), 令单位根 $\omega = \sqrt[3]{b^3 - a}$, 把域内所有数表示为 $\omega^0, \omega^1, \omega^2$ 的线性组合, 然后答案即为 $(b - \omega)^{\frac{p^2+p+1}{3}}$。

证明:

引理 4.38 $\omega^{p-1} = (t^3 - a)^{\frac{p-1}{3}} = \epsilon \pmod p$

引理 4.39 $\epsilon(1 + \epsilon) = (\epsilon - \epsilon^3)/(1 - \epsilon) = -1 \pmod p$

$$\begin{aligned}
(b-\omega)^{p^2+p+1} &= (b-\omega)(b-\omega)^{p^2}(b-\omega)^p \\
&\equiv (b-\omega)(b^{p^2} - \omega^{p^2})(b^p - \omega^p) \\
&\equiv (b-\omega)(b^2 - b\omega\epsilon - b\omega\epsilon^2 + \omega^2) \\
&= (b-\omega)(b^2 + b\omega + \omega^2) \\
&= b^3 - \omega^3 \\
&= a \pmod p
\end{aligned}$$

[35][数论] 二次剩余及计算方法 http://blog.miskcoo.com/2014/08/quadratic-residue
[36]二次同余方程的解 https://blog.csdn.net/acdreamers/article/details/10182281
[37]二次同余方程模合数的一般解法 https://blog.csdn.net/Quack_quack/article/details/50189111

上述内容参考了 skywalkert[38] 与 hzq84621[39]的博客。

4.12.4 高次剩余

高次剩余仍然可以使用上文的随机化算法解决, 即寻找 b 满足 $(b^n-a)^{\frac{p-1}{n}}\neq 1$, 记 $\omega=\sqrt[n]{b^n-a}$, 把数域扩充为 $\omega^0,\omega^1,\cdots,\omega^{n-1}$ 的线性组合, 答案为 $(b-\omega)^{\frac{1}{n}\sum_{i=0}^{n-1}p^i}$。然后解可乘以单位 n 次根的幂以生成整个解集, 单位 n 次根可以使用求原根得到。期望尝试次数 $n^{n-1}\lg p$。

这里介绍另一种方法: 若模 p 有原根 g, 令 $x=g^t$, 有 $g^{nt}\equiv a\pmod p$, 变形得 $(g^n)^t\equiv a\pmod p$, 转换为离散对数问题求解, 具体方法参见 4.10.1 节所述内容。

上述内容参考了静听风吟。[40]的博客。

4.12.5 Tonelli-Shanks 算法

留坑待补。

4.13 类欧几里得算法

已知 k_1,k_2,a,b,c,n, 求

$$F(k_1,k_2,a,b,c,n)=\sum_{x=0}^{n}x^{k_1}\left\lfloor\frac{ax+b}{c}\right\rfloor^{k_2}$$

其中 k_1,k_2 较小。

以下除法操作均指向下取整除法, λ 为易求得的常数。讨论多种情况的处理方法:

- 若 $an+b<c\wedge k_2\neq 0$, 直接返回 0。
- 若 $k_2=0\vee a=0$, 答案为 $\lambda\sum_{x=0}^{n}x^{k_1}$。利用 9.5.5所述方法求解。
- $a\geq c\vee b\geq c$:

  令 $F(k_1,k_2,a,b,c,n)=$
  $\sum_{x=0}^{n}x^{k_1}((a/c)x+(b/c)+((a \bmod c)\cdot x+(b \bmod c))/c)^{k_2}$, 对指数为 k_2 的部分进行多项式展开, 可以拆出类似 $\sum\lambda F$ 的形式, 递归求解。
- 否则 $a<c\wedge b<c\wedge k_2\neq 0$, 使用类似求期望的技巧, 枚举指数为 k_2 的幂并差分, 记

[38]寻找模素数意义下的二次剩余与三次剩余 https://blog.csdn.net/skywalkert/article/details/52591343

[39]二次剩余和三次剩余相关 http://hzq84621.is-programmer.com/posts/208648.html

[40]数学:二次剩余与 n 次剩余 https://www.cnblogs.com/aininot260/p/9709283.html

$m=(an+b)/c$, 有

$$\begin{aligned}F(k_1,k_2,a,b,c,n) &= \sum_{i=0}^{m-1}\left(((i+1)^{k_2}-i^{k_2})\sum_{x=0}^{n}x^{k_1}[(ax+b)/c>i]\right)\\ &= \sum_{i=0}^{m-1}\left(((i+1)^{k_2}-i^{k_2})\sum_{x=0}^{n}x^{k_1}[x>(ic+c-b-1)/a]\right)\\ &= \sum_{i=0}^{m-1}((i+1)^{k_2}-i^{k_2})\cdot\sum_{x=0}^{n}x^{k_1}\\ &\quad -\sum_{i=0}^{m-1}\left(((i+1)^{k_2}-i^{k_2})\sum_{x=0}^{(ic+c-b-1)/a}x^{k_1}\right)\end{aligned}$$

前一部分可以插值求出, 后一部分把多项式 $\sum_{i=0}^{n}i^k=\sum_{i=0}^{k+1}c_i\cdot n^i$ 展开, 化为 $\sum\lambda F$ 的形式递归求解。

实现时可以令函数返回 $g(n,a,b,c)=R^{k_1\cdot k_2}$ 存储所有需要的值, 避免重复计算。递归层数为 $O(\lg n)$。

模板(LOJ#138):

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>
const int mod = 1000000007;
int add(int a, int b) {
    a += b;
    return a < mod ? a : a - mod;
}
int sub(int a, int b) {
    a -= b;
    return a >= 0 ? a : a + mod;
}
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
Int64 powm(Int64 x, int k) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * x % mod;
        k >>= 1, x = x * x % mod;
    }
    return res;
}
Int64 inv(Int64 x) {
    return powm(x, mod - 2);
}
const int maxp = 15;
```

```
struct Poly {
    int k, fac[maxp];
    void construct(int* pv, int cv) {
        int base[maxp];
        for(int i = 1; i <= cv; ++i) {
            Int64 cur = 1;
            for(int j = 1; j <= cv; ++j) {
                if(j == i)
                    continue;
                cur = cur * (i - j) % mod;
            }
            base[i] = pv[i] * inv(cur) % mod;
        }
        k = cv;
        int L[maxp][maxp], R[maxp][maxp];
        L[0][0] = R[k + 1][0] = 1;
        for(int i = 1; i <= k; ++i) {
            L[i][0] = 0;
            for(int j = 1; j <= i; ++j)
                L[i][j] = L[i - 1][j - 1];
            for(int j = 0; j < i; ++j)
                L[i][j] = (L[i][j] -
                           asInt64(i) * L[i - 1][j]) %
                    mod;
        }
        for(int i = k; i >= 1; --i) {
            int c = k - i + 1;
            R[i][0] = 0;
            for(int j = 1; j <= c; ++j)
                R[i][j] = R[i + 1][j - 1];
            for(int j = 0; j < c; ++j)
                R[i][j] = (R[i][j] -
                           asInt64(i) * R[i + 1][j]) %
                    mod;
        }
        for(int i = 0; i < k; ++i)
            fac[i] = 0;
        for(int i = 1; i <= k; ++i) {
            int lc = i - 1, rc = k - i;
            Int64 bv = base[i];
            for(int j = 0; j <= lc; ++j)
                for(int k = 0; k <= rc; ++k) {
                    fac[j + k] =
                        (fac[j + k] +
                         bv * L[i - 1][j] % mod *
                             R[i + 1][k]) %
                        mod;
                }
        }
        for(int i = 0; i < k; ++i)
```

```
            if(fac[i] < 0)
                fac[i] += mod;
    }
    int operator()(int x) const {
        Int64 res = 0;
        for(int i = k - 1; i >= 0; --i)
            res = (res * x + fac[i]) % mod;
        return res;
    }
} p[maxp], d[maxp];
void prePoly(int k) {
    int end = k + 2;
    int pv[maxp][maxp], spv[maxp];
    for(int i = 1; i <= end; ++i) {
        pv[i][0] = 1;
        for(int j = 1; j <= k; ++j)
            pv[i][j] = asInt64(pv[i][j - 1]) * i % mod;
    }
    for(int i = 0; i <= k; ++i) {
        spv[0] = (i == 0);
        int end = i + 2;
        for(int j = 1; j <= end; ++j)
            spv[j] = add(spv[j - 1], pv[j][i]);
        p[i].construct(spv, end);
        for(int j = 1; j < end; ++j)
            spv[j] = sub(pv[j + 1][i], pv[j][i]);
        d[i].construct(spv, end - 1);
    }
}
struct PolyExpr {
    int a, b, c;
    PolyExpr(int a, int b, int c) : a(a), b(b), c(c) {}
    PolyExpr operator*(const PolyExpr& rhs) const {
        return PolyExpr(a + rhs.a, b + rhs.b,
                        asInt64(c) * rhs.c % mod);
    }
    bool operator<(const PolyExpr& rhs) const {
        return a != rhs.a ? a < rhs.a : b < rhs.b;
    }
    bool merge(const PolyExpr& rhs) {
        if(a == rhs.a && b == rhs.b) {
            c = add(c, rhs.c);
            return true;
        }
        return false;
    }
};
typedef std::vector<PolyExpr> SPE;
void prePolyExpr(std::vector<SPE>& expr, int k, int a,
                 int b) {
```

```
    SPE base;
    base.push_back(PolyExpr(1, 0, a));
    base.push_back(PolyExpr(0, 1, 1));
    base.push_back(PolyExpr(0, 0, b));
    expr.resize(k + 1);
    expr[0].push_back(PolyExpr(0, 0, 1));
    for(int i = 1; i <= k; ++i) {
        SPE cur, &last = expr[i - 1];
        cur.reserve(base.size() * last.size());
        for(int j = 0; j < base.size(); ++j)
            for(int k = 0; k < last.size(); ++k)
                cur.push_back(base[j] * last[k]);
        std::sort(cur.begin(), cur.end());
        PolyExpr cexp(0, 0, 0);
        SPE& post = expr[i];
        for(int j = 0; j < cur.size(); ++j) {
            if(!cexp.merge(cur[j])) {
                if(cexp.c)
                    post.push_back(cexp);
                cexp = cur[j];
            }
        }
        if(cexp.c)
            post.push_back(cexp);
    }
}
struct Mat {
    int A[15][15];
    int* operator[](int x) {
        return A[x];
    }
};
int mp;
Mat f(int n, int a, int b, int c) {
    Mat res;
    Int64 end = asInt64(a) * n + b;
    for(int i = 0; i <= mp; ++i)
        res[i][0] = p[i](n);
    if(end < c) {
        for(int i = 0; i <= mp; ++i) {
            for(int j = 1; i + j <= mp; ++j)
                res[i][j] = 0;
        }
    } else if(a == 0) {
        Int64 div = b / c;
        for(int i = 0; i <= mp; ++i) {
            Int64 k = res[i][0];
            for(int j = 1; i + j <= mp; ++j) {
                k = k * div % mod;
                res[i][j] = k;
```

```
            }
        }
    } else if(a >= c || b >= c) {
        int adc = a / c, amc = a % c, bdc = b / c,
            bmc = b % c;
        std::vector<SPE> expr;
        prePolyExpr(expr, mp, adc, bdc);
        Mat info = f(n, amc, bmc, c);
        for(int i = 0; i <= mp; ++i) {
            for(int j = 1; i + j <= mp; ++j) {
                int sum = 0;
                for(int k = 0; k < expr[j].size();
                    ++k) {
                    PolyExpr cexp = expr[j][k];
                    sum = (sum +
                           asInt64(cexp.c) *
                               info[i + cexp.a]
                                   [cexp.b]) %
                          mod;
                }
                res[i][j] = sum;
            }
        }
    } else {
        int m = end / c, nb = sub(sub(c, b), 1);
        Mat info = f(m - 1, c, nb, a);
        for(int i = 0; i <= mp; ++i) {
            Int64 l = res[i][0];
            for(int j = 1; i + j <= mp; ++j) {
                l = l * m % mod;
                int r = 0;
                Poly &p1 = p[i], &p2 = d[j];
                for(int x = 0; x < p1.k; ++x)
                    for(int y = 0; y < p2.k; ++y) {
                        r = (r +
                             asInt64(p1.fac[x]) *
                                 p2.fac[y] % mod *
                                 info[y][x]) %
                            mod;
                    }
                res[i][j] = sub(l, r);
            }
        }
    }
    return res;
}
int main() {
    prePoly(10);
    int t;
    scanf("%d", &t);
```

```
    while(t--) {
        int n, a, b, c, k1, k2;
230     scanf("%d%d%d%d%d%d", &n, &a, &b, &c, &k1,
              &k2);
        mp = k1 + k2;
        Mat res = f(n, a, b, c);
        printf("%d\n", res[k1][k2]);
    }
    return 0;
}
```

上述内容参考了 fjzzq2002 的博客[41]。

WC2019 营员交流中有营员(忘了是谁)提出了“万能欧几里得”算法, 可以简单解决类欧几里得问题。

[41]类欧几里得算法 https://www.cnblogs.com/zzqsblog/p/8904010.html

Chapter 5

集合论 群论

5.1 集合论定理 190
5.1.1 模意义统计方案 191
5.2 拉格朗日定理 191
5.3 置换群 191
5.3.1 Burnside 引理 191
5.3.2 Pólya 定理 192
5.3.3 常见题型 192
5.3.4 完全图染色问题 197
5.3.5 置换的群性质 199

5.1 集合论定理

定理 5.1 (对称差)

$$A \oplus B = (A - B) \cup (B - A) = A \cup B - A \cap B$$

定理 5.2 (De Morgan's Laws)

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

定理 5.3 (Inclusion-exclusion Principle)

$$\left| \bigcup_{i=1}^{n} A_i \right| = \sum_{\emptyset \neq J \subseteq \{1,2,\cdots,n\}} (-1)^{|J|-1} \left| \bigcap_{j \in J} A_j \right|$$

容斥原理用来求集合并的大小, 为了求集合交的大小, 可以使用补集转换思想, 由定理 5.2与 5.3可得

定理 5.4

$$\left| \bigcap_{i=1}^{n} A_i \right| = \left| \overline{\bigcup_{i=1}^{n} \overline{A_i}} \right| = |U| + \sum_{\emptyset \neq J \subseteq \{1,2,\cdots,n\}} (-1)^{|J|} \left| \bigcap_{j \in J} \overline{A_j} \right|$$

以上内容参考了 Wikipedia-EN[1]以及国家集训队 2013 论文集《浅谈容斥原理》。

5.1.1 模意义统计方案

若要求恰好满足 k 个条件的方案数，且这些条件是等价的。考虑使用至少满足 k 个条件的方案数容斥求出，后者由于限制条件较松，很容易使用 NTT 等方法得到。

记 $g(x)$ 为至少满足 x 个条件的方案数，那么 $g(i), i \geq k$ 构成的每种方案都对应 $\binom{i}{k}$ 种满足 k 个条件的方案。由容斥可得 $ans = \sum_{i=k}^{n} (-1)^{i-k} \binom{i}{k} g(i)$。

5.2 拉格朗日定理

定理 5.5 (Lagrange's Theorem) 若 $(S, \oplus)$ 是一个有限群，$(S', \oplus)$ 是 $(S, \oplus)$ 的子群，则 $|S'|$ 是 $|S|$ 的约数。

证明留坑待补。

5.3 置换群

置换是从 $[1, n]$ 到 $[1, n]$ 的双射。
一个置换可以分解为多个内部循环，计算循环相关信息的方法为：
枚举每一个节点：

1. 若该节点已被访问，则跳过；
2. 顺着该节点对应的目标节点不断跳跃，给访问到的点打标记，直至跳跃到已访问点（即出发点）为止。
3. 这个环就是一个循环。

定理 5.6 若一个置换内部有 n 个循环，长度分别为 $l_1, l_2, \cdots, l_n$，则该置换的循环节长度为 $lcm(l_1, l_2, \cdots, l_n)$。

不动点 若一个状态 S 经由置换 f 置换后的状态与原状态相同，则状态 S 为 f 的不动点。

等价关系 对于一个置换集合 F，若状态 S 能经由 F 中的置换变为状态 S'，则称 S 与 S' 等价。

等价类 满足等价关系的状态属于同一等价类。

5.3.1 Burnside 引理

引理 5.7 (Burnside's Lemma) 等价类数目为置换群 G 中所有置换的不动点数目的平均值。

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|$$

上述定理证明留坑待补。

[1]Inclusion–exclusion principle - Wikipedia
https://en.wikipedia.org/wiki/Inclusion%E2%80%93exclusion_principle
De Morgan's laws - Wikipedia https://en.wikipedia.org/wiki/De_Morgan%27s_laws

5.3.2 Pólya 定理

定理 5.8 (Pólya Enumeration Theorem) 若对每一个节点进行 m 染色，置换 g 有 $c(g)$ 个循环，则染色方案等价类数目为 $\frac{1}{|G|}\sum_{g\in G} m^{c(g)}$。

证明：要使状态为不动点，必须让同循环内的节点同颜色，循环的颜色选择独立，根据乘法原理可知 $|X^g| = m^{c(g)}$。

以上内容参考了 QAQqwe 的博客[2]与 Wikipedia-EN[3]。

5.3.3 常见题型

题型来自 My_ACM_Dream 的博客[4]。

正方形旋转 n*n 正方形染色：

- 旋转 $0°$，循环数 n^2。
- 旋转 $90°/270°$，若 n 为偶数，循环数 $\frac{n^2}{4}$；若 n 为奇数，循环数 $\frac{n^2-1}{4}+1$。
- 旋转 $180°$，若 n 为偶数，循环数 $\frac{n^2}{2}$；若 n 为奇数，循环数 $\frac{n^2-1}{2}+1$。

奇偶循环数不同的原因是因为当 n 为奇数时中间的点自成一个循环。

正方形反射 (对称)

	对角反射	对边中点反射
n 为奇数	$\frac{n^2-n}{2}+n$	$\frac{n^2-n}{2}+n$
n 为偶数	$\frac{n^2-n}{2}+n$	$\frac{n^2}{2}$

环形旋转 对于一个有 n 个点的环，旋转 i 个点的置换的循环数为 (n,i)。

证明：每个循环的长度为 $\frac{[n,i]}{i} = \frac{n}{(n,i)}$，循环数为 (n,i)。

环形对称翻转

- n 为奇数：只有 n 种置换(以一点一边中点为对称轴)，循环数为 $[\frac{n}{2}]+1$。
- n 为偶数，按照对称轴分类：
  - 边边中点：$\frac{n}{2}$ 种，循环数为 $\frac{n}{2}$。
  - 点点：$\frac{n}{2}$ 种，循环数为 $\frac{n}{2}+1$。

[2]Burnside 引理与 Polya 定理 https://blog.csdn.net/liangzhaoyang1/article/details/72639208

[3]Burnside's lemma - Wikipedia https://en.wikipedia.org/wiki/Burnside%27s_lemma
Pólya enumeration theorem - Wikipedia https://en.wikipedia.org/wiki/P%C3%B3lya_enumeration_theorem

[4]polya|burnside 定理的一些总结
https://blog.csdn.net/My_ACM_Dream/article/details/45312365

正方体旋转 注意是**棱边**置换。

- 自身不变, 置换 1 种, 循环 12 个, 长度 1;
- 以对面中心为轴, 旋转角为 $90°, 180°, 270°$, 轴有 3 种选择, 共 9 种置换。
  - $90°/270°$:循环 3 个, 长度 4。
  - $180°$:循环 6 个, 长度 2。
- 以对边中点为轴, 旋转角为 $180°$, 有 6 对边, 置换数为 6, 有 5 个长度为 2 的循环和 2 个长度为 1 的循环。
- 以对顶点为轴, 旋转角为 $120°, 240°$, 有 4 对点, 置换数为 8, 均有 4 个长度为 3 的循环。

总置换数 24。

例题:UVA10601 Cubes

暴力(矩阵变换 + 枚举排列检查不动点):

```
#include <algorithm>
#include <cmath>
#include <cstdio>
typedef double FT;
const FT deg2Rad = acos(-1.0) / 180.0;
struct Vec {
    FT x, y, z;
    Vec() {}
    Vec(int id)
        : x(id & 1 ? 1.0 : -1.0),
          y(id & 2 ? 1.0 : -1.0),
          z(id & 4 ? 1.0 : -1.0) {}
    Vec(FT x, FT y, FT z) : x(x), y(y), z(z) {}
    void normalize() {
        FT dis = sqrt(x * x + y * y + z * z);
        x /= dis, y /= dis, z /= dis;
    }
    FT operator[](int idx) const {
        if(idx == 0)
            return x;
        if(idx == 1)
            return y;
        return z;
    }
    FT& operator[](int idx) {
        if(idx == 0)
            return x;
        if(idx == 1)
            return y;
        return z;
    }
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y, z - rhs.z);
```

```
    }
    int getId() const {
        return (x > 0.0 ? 1 : 0) | (y > 0.0 ? 2 : 0) |
            (z > 0.0 ? 4 : 0);
    }
};
Vec cross(const Vec& a, const Vec& b) {
    return Vec(a.y * b.z - a.z * b.y,
               a.z * b.x - a.x * b.z,
               a.x * b.y - a.y * b.x);
}
FT dot(const Vec& a, const Vec& b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
struct Mat {
    Vec A[3];
    Vec operator[](int idx) const {
        return A[idx];
    }
    Vec& operator[](int idx) {
        return A[idx];
    }
    Vec operator()(const Vec& rhs) const {
        return Vec(dot(A[0], rhs), dot(A[1], rhs),
                   dot(A[2], rhs));
    }
};
Mat genCoordSystem(Vec axis) {
    Vec base(1.0, 2.0, 3.0);
    Mat res;
    res[0] = cross(base, axis);
    res[1] = cross(res[0], axis);
    res[2] = axis;
    for(int i = 0; i < 3; ++i)
        res[i].normalize();
    return res;
}
Mat getInv(const Mat& rhs) {
    Mat res;
    for(int i = 0; i < 3; ++i)
        for(int j = 0; j < 3; ++j)
            res[i][j] = rhs[j][i];
    return res;
}
int lut[24][12], tid[8], link[8][8];
void rotate(int id, const Vec& axis, FT deg) {
    Mat trans = genCoordSystem(axis),
        inv = getInv(trans), rot;
    FT rad = deg * deg2Rad;
    rot[0] = Vec(cos(rad), -sin(rad), 0.0);
```

```
    rot[1] = Vec(sin(rad), cos(rad), 0.0);
    rot[2] = Vec(0.0, 0.0, 1.0);
    for(int i = 0; i < 8; ++i) {
        Vec tp = inv(rot(trans(Vec(i))));
        tid[i] = tp.getId();
    }
    for(int i = 0; i < 8; ++i) {
        for(int j = 1; j <= 4; j <<= 1)
            if(i & j) {
                int k = i ^ j, tu = tid[i],
                    tv = tid[k];
                if(tu < tv)
                    std::swap(tu, tv);
                lut[id][link[i][k]] = link[tu][tv];
            }
    }
}
void genTable() {
    int eid = 0;
    for(int i = 0; i < 8; ++i)
        for(int j = 1; j <= 4; j <<= 1)
            if(i & j)
                link[i][i ^ j] = eid++;
    for(int i = 0; i < 12; ++i)
        lut[0][i] = i;
    int id = 0;
    for(int i = 0; i < 3; ++i)
        for(int j = 90; j <= 270; j += 90)
            rotate(++id, Vec(i == 0, i == 1, i == 2),
                   j);
    for(int i = 0; i < 3; ++i) {
        Vec base(i != 0, i != 1, i != 2);
        rotate(++id, base, 180.0);
        for(int j = 0; j < 3; ++j)
            if(base[j] == 1) {
                base[j] = -1;
                break;
            }
        rotate(++id, base, 180.0);
    }
    for(int i = 0; i < 8; ++i)
        if(i < (i ^ 7)) {
            Vec base = Vec(i) - Vec(i ^ 7);
            rotate(++id, base, 120.0);
            rotate(++id, base, 240.0);
        }
}
int col[12];
int foo() {
    for(int i = 0; i < 12; ++i)
```

```
        scanf("%d", &col[i]);
    std::sort(col, col + 12);
    int cnt = 0;
    do {
        for(int i = 0; i < 24; ++i) {
            bool flag = true;
            for(int j = 0; j < 12; ++j)
                if(col[lut[i][j]] != col[j]) {
                    flag = false;
                    break;
                }
            cnt += flag;
        }
    } while(std::next_permutation(col, col + 12));
    return cnt / 24;
}
int main() {
    genTable();
    int t;
    scanf("%d", &t);
    while(t--)
        printf("%d\n", foo());
    return 0;
}
```

正解:首先统计每种颜色的数量,然后分类讨论,每种情况的贡献为置换数 * 循环方案数。若一种置换的所有循环长度相等,则使用排列组合知识求解(solve 函数);否则枚举短循环的颜色,令剩余循环长度相等,对剩余颜色和循环调用 solve 求解。

```
#include <cstdio>
#include <cstring>
int A[6], fac[13];
int solve(int len, int base = 12) {
    int ans = fac[base / len];
    for(int i = 0; i < 6; ++i) {
        int c = A[i] / len, d = A[i] % len;
        if(d)
            return 0;
        ans /= fac[c];
    }
    return ans;
}
int foo() {
    memset(A, 0, sizeof(A));
    for(int i = 0; i < 12; ++i) {
        int col;
        scanf("%d", &col);
        ++A[col - 1];
    }
    int ans = solve(1) + 3 * solve(2) + 8 * solve(3) +
        6 * solve(4);
```

```
    for(int i = 0; i < 6; ++i)
        for(int j = 0; j < 6; ++j) {
            --A[i], --A[j];
            if(A[i] >= 0 && A[j] >= 0)
                ans += 6 * solve(2, 10);
            ++A[i], ++A[j];
        }
    return ans / 24;
}
int main() {
    fac[0] = 1;
    for(int i = 1; i <= 12; ++i)
        fac[i] = fac[i - 1] * i;
    int t;
    scanf("%d", &t);
    while(t--)
        printf("%d\n", foo());
    return 0;
}
```

n 较小

- 颜色不限：裸 Polya 解决。
- 颜色限制：裸 Burnside 解决。

环形旋转且 n 较大 枚举循环数(即 $d=(n,i)$)，利用欧拉函数与容斥解决。

考虑枚举 n 的因子 $k=(n,i)$，记有 k 个循环的不动点个数为 $f(k)$，那么根据定理 5.7，等价类数目为

$$\frac{1}{n}\sum_{k|n} f(k) \sum_{k|i,i<n} [(\frac{n}{k},\frac{i}{k})=1] = \frac{1}{n}\sum_{k|n} f(k)\varphi(\frac{n}{k})$$

该内容参考了国家集训队 2013 论文集高胜寒的《浅谈环状计数问题》。

有染色限制 使用 dp 与矩阵快速幂解决。

5.3.4 完全图染色问题

有一个由 n 个点($n \le 60$)组成的完全图，每条边可被染成 m 种颜色中的一种，求本质不同的染色图数。

若使用 Pólya 定理解决，可以发现 $|G|$ 的规模达到 $N!$，显然枚举置换过不了。

考虑将点置换转换为边置换，即 $(u,v)\to(P_u,P_v)$。

然后将其分为 2 种情况：

- 若点置换内有一个循环长度为 L，则对应边置换内有 $\left[\frac{L}{2}\right]$ 个循环。

  证明：所有端点在点置换循环上距离(定义为最短路)相等的边构成一个等价类，而等价类数目有 $\left[\frac{L}{2}\right]$ 个。

- 若点置换内有两个循环长度分别为 L_1, L_2, 则对应边置换内有 (L_1, L_2) 个循环。

  证明: 使两个循环都置换到原状态需要 $[L_1, L_2]$ 次, 即每个循环的大小。而总边数为 L_1L_2, 因此循环数为 (L_1, L_2)。

接下来考虑枚举点置换的循环, 记这些循环的长度为 $L_1, L_2, \cdots, L_k$。首先 DFS 枚举保证 $L_1 \geq L_2 \geq \cdots \geq L_k > 0$, 即求 n 的划分方案。

然后对于某个划分可以由上文方法求出对应置换的循环个数, 接着考虑该划分对应的置换数。总置换数为 $N!$, 将 n 个点分配给这些循环, 去掉内部排列导致的方案重复, 要除以 $\prod_{i=1}^{k} L_i!$。对于每一个循环, 一旦首位固定, 其它位的排列顺序不同也会构造出不同的置换, 方案数再乘以 $\prod_{i=1}^{k} (L_i - 1)!$。实际上只需除以 $\prod_{i=1}^{k} L_i$, 可以理解为一个固定的有向环可以任意指定起点。等长循环的排列顺序会造成方案重复, 记长度为 L 的循环数为 c_L, 再除以 $\prod_{i=1}^{n} c_i!$。最终置换数为

$$\frac{N!}{\prod_{i=1}^{k} L_i \prod_{i=1}^{n} c_i!}$$

分子的 $N!$ 与 Burnside 引理的系数相消。

代码([HNOI2009] 图的同构记数):

```
#include <algorithm>
#include <cstdio>
const int mod = 997, phi = mod - 1, size = 65;
int gcd(int a, int b) {
    return b ? gcd(b, a % b) : a;
}
int inv[mod], p2[mod], L[size], c[size], ccnt = 0,
                                          ans = 0;
void DFS(int n, int maxl, int a, int b) {
    if(n) {
        for(int i = 1; i <= maxl; ++i) {
            int nb = (b + (i >> 1)) % phi;
            for(int j = 1; j <= ccnt; ++j)
                nb = (nb + gcd(L[j], i)) % phi;
            L[++ccnt] = i, ++c[i];
            int na = a * c[i] * i % mod;
            DFS(n - i, std::min(n - i, i), na, nb);
            --ccnt, --c[i];
        }
    } else
        ans = (ans + inv[a] * p2[b]) % mod;
}
int main() {
    inv[1] = 1;
    for(int i = 2; i < mod; ++i)
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
    p2[0] = 1;
    for(int i = 1; i < phi; ++i)
```

```
        p2[i] = (p2[i − 1] << 1) % mod;
    int n;
    scanf("%d", &n);
    DFS(n, n, 1, 0);
    printf("%d\n", ans);
    return 0;
}
```

上述内容参考了陈瑜希的论文《Pólya 计数法的应用》。

5.3.5 置换的群性质

注意单个置换可以生成群, 且满足结合律, 因此。。。

- 置换快速幂

  置换快速幂可以在 $O(n)$ 而不是 $O(n \lg n)$ 内解决。考虑求出置换的每个循环, 循环内的幂可以 $O(1)$ 找到对应位置。

- 置换求离散对数(BSGS)

  利用循环的性质可以避开 BSGS。当对应位置的元素不在同一个循环内时无解, 否则可以根据环上位置差列出模线性方程组。

例题:FCS2019 Day6 blue

给定一个操作 $f(S)$, 这个操作每次将串 S 循环左移 a 位, 再对串的每个字母做 b 次置换。求 S 至少做多少次操作才能变成 T, 或者无法变成 T。

首先 b 次置换可以使用置换快速幂变为 1 次置换。

我原先的做法:求出置换和循环左移的循环长度, 然后暴力。

注意到 $f(S)$ 是双射, 也有逆变换, 因此可以将题目表示为求离散对数 $f^x(S) = T$, 然后转换为 BSGS 解决。当然还要提前计算出 $f(S)$ 的周期, 然后就可以将复杂度中周期的因子降为根号。

$f(S)$ 由多个子置换组成, 它的生成群的阶是这些子置换循环长度的 lcm。这说明这个置换可以分解为子问题解决, 类似于 4.10.4节所述的 Pohlig–Hellman 算法。考虑枚举 S 与 T 的起始位置之差, 这个时候 S 和 T 对应位置的字母是匹配的。要求对应字母在同一个环内且同环字母对的环上位置差相同。暴力枚举起始位置匹配不可行, 考虑将约束表达为字符串然后做 KMP。一个关键字是环 id, 另一个关键字是自身与串前面的同环字母的位置差(考虑 S 有同环位置 a, b, T 有同环位置 c, d, $a - c = b - d \Rightarrow b - a = d - c$)。使用 KMP 求出所有匹配位置, 同时记录已匹配串的环上位置差。记起始位置差为 A, 环上位置差为 B_i, 环长为 C_i, 可以将原问题表示为 $ax \equiv A \pmod{n}, bx \equiv B_i \pmod{C_i}$, 使用 ExCRT 解决。

Chapter 6

组合数学

6.1 盒子放球问题总结 200
6.2 Catalan 数 201
6.2.1 性质 201
6.2.2 常见应用 202
6.2.3 扩展 203
6.3 Stirling 数 206
6.3.1 第一类 Stirling 数 206
6.3.2 第二类 Stirling 数 207
6.4 贝尔数 207
6.4.1 因子分解应用 208
6.4.2 贝尔数计算 208
6.5 Lucas/ExLucas 210
6.5.1 Lucas 定理 210
6.5.2 ExLucas 211
6.6 康托展开 212
6.7 其它公式与定理 212
6.7.1 常用公式 212
6.7.2 Raney 引理 213

6.1 盒子放球问题总结

该系列问题叫做 “The twentyfold way”, 以下内容参考 Wikipedia-EN [1]、自为风月马前卒的博客 [2]以及洛谷日报 chengni 的文章[3]。这里仅摘取 8 种常见情况。

假设将 n 个球放入 m 个盒子中。

6.1.0.1 球异,可空

- 盒异:视为染色问题, m^n

[1]Twelvefold way - Wikipedia https://en.wikipedia.org/wiki/Twelvefold_way
[2]浅谈"n 个球" 和"m 个盒子" 之间的乱伦关系 https://www.cnblogs.com/zwfymqz/p/9724918.html
[3]当小球遇上盒子 https://www.luogu.org/blog/chengni5673/dang-xiao-qiu-yu-shang-he-zi

- 盒同:枚举非空盒子数,转化为球异非空盒同问题,$\sum_{i=0}^{m} \left\{ {n \atop i} \right\}$

6.1.0.2 球异,非空

- 盒异:计算出盒同的所有情况后给每个盒子编号,$m! \left\{ {n \atop m} \right\}$
- 盒同:即第二类斯特林数,$\left\{ {n \atop m} \right\}$

6.1.0.3 球同,可空

- 盒异:使用隔板法,强制给每个盒子额外一个球,$\binom{n+m-1}{m-1}$
- 盒同:实际上要求的是长度为 m 的有序数列个数,数列的元素和为 n。
  记 $dp[n][m]$ 为划分方案数,有

$$\begin{aligned} dp[0][k] &= dp[k][1] = 1 \\ dp[i][j] &= dp[i-j][j] + dp[i][j-1] \end{aligned}$$

  状态转移方程解释:考虑 j 个盒子中是否有空盒

  - 有空盒:新增一个空盒,贡献 $dp[i][j-1]$
  - 无空盒:向每个盒子放入一个球使其非空,贡献 $dp[i-j][j]$

6.1.0.4 球同,非空

- 盒异:使用隔板法,$\binom{n-1}{m-1}$
- 盒同:强制给每个盒子一个球后转化为可空,$dp[n-m][m]$

6.2 Catalan 数

6.2.1 性质

Catalan 数是组合数学中的常见数列 [4],其前几项为

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, \cdots$$

Catalan 数(记为 C_n)满足如下关系:

$$C_0 = C_1 = 1 \tag{6.1}$$

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \tag{6.2}$$

$$C_n = \frac{4n-2}{n+1} C_{n-1} \tag{6.3}$$

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n} \tag{6.4}$$

$$C_n = \prod_{k=2}^{n} \frac{n+k}{k} \tag{6.5}$$

[4] A000108 - OEIS http://oeis.org/A000108

根据 Striling 近似公式

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

可得

$$C_n \sim \frac{4^n}{\sqrt{\pi} n^{\frac{3}{2}}}$$

定理 6.1 C_n 为奇数当且仅当 $n = 2^k - 1$。

6.2.2 常见应用

6.2.2.1 括号序列,出栈序列,网格行走

括号序列 给定 $2n$ 个位置填上左右括号使括号匹配(对于每一个位置之前的左括号必须不少于右括号)。

出栈序列 求将 n 个元素入栈一次(限制顺序)并出栈一次(不限制顺序)的方案数(对于每一次操作都要保证栈不出现下溢,即任意时刻入栈元素不少于出栈元素)。

网格行走 在一个 $n * n$ 的网格内从左下角移动到右上角,纵坐标必须不小于横坐标,求方案数。

分析 这三个问题是同构的,都满足操作数为 $2n$ 且限制任意时刻操作 A 的数目不少于操作 B 的数目。它们的答案都是 C_n,以括号序列问题为例,通过等式 6.2理解:将括号序列看做由一个可分割的序列加上一个不可分割的序列(即最外层有一对配对括号)得来,左边为 $n_1 + 1$ 对,右边为 n_2 对,满足 $n_1 + n_2 = n - 1$,这种方案的贡献为 $C_{n_1} C_{n_2}$。

6.2.2.2 二叉树构型计数

有 n 个节点的二叉树 通过等式 6.2理解: 枚举左右子树大小, 满足左右子树节点数为 $n - 1$。

有 $n + 1$ 个叶子节点的满二叉树 满二叉树去掉叶子节点后是一棵二叉树,并且在满二叉树中叶子节点比非叶子节点多一个。

6.2.2.3 阶梯填充

用 n 个长方形填充 $n * n$ 的阶梯的方案数为 C_n。

证明: 填充一个以直角顶点与某阶梯顶点为对顶点的长方形, 使其分为大小为 $n_1 * n_1, n_2 * n_2$ 的两个小阶梯,满足 $n_1 + n_2 = n - 1$。按大小把长方形数量份额分配给两个子问题($n * n$ 的阶梯至少需要 n 个长方形,所以必须严格按照大小比例分配)。该分析满足等式6.2。

6.2.2.4 凸包分割

将 $n + 2$ 个顶点的凸包分为三角形的方案数为 C_n。

证明留坑待补。

6.2.2.5 圆上点连线

将圆上的 $2n$ 个点两两配对连线, 所连 n 条线段不相交的方案数为 C_n。
证明留坑待补。
上述内容参考了 Wikipedia-EN[5]。

6.2.3 扩展

有 n 个 1 和 m 个-1(n>m), 这些数排成一个序列, 需要满足序列每个位置上的前缀和 >0, 求序列方案数。

该问题可转换为在方格纸上从 $(0,0)$ 走到 (n,m), 且仅在对角线 $x=y$ 下方走的路径数。

考虑第一步走到哪个点:

- 走到 $(0,1)$, 显然该走法的后继路径不合法, 方案数为 $\binom{n+m-1}{n}$。
- 走到 $(1,0)$, 若该走法的后继路径不合法, 则该路径的从开始到第一次触碰到对角线的路径沿对角线 $x=y$ 对称后可使其一一映射到情况 1 的每条路径。

因此答案为无限制路径数-情况 1 方案数-情况 2 方案数 $=\binom{n+m}{n}-2\cdot\binom{n+m-1}{n}=\binom{n+m-1}{m}-\binom{n+m-1}{m-1}$

6.2.3.1 变形 a

$n\geq m$, 满足前缀和 ≥ 0。
把路径终点定为 $(n+1,m)$, 可将此问题转化为上一个问题(因为第一步必走 $(1,0)$, 等价于将起点和终点右移一位, 或者对角线左移), 答案为 $\binom{n+m}{m}-\binom{n+m}{m-1}$。

6.2.3.2 变形 b

$n\geq m-r$, 且满足前缀和 $\geq -r, r\in\mathbb{Z}$。
画图可知求的是从 $(r,0)$ 走到 $(r+n,m)$ 且不穿过对称轴 $x=y$ 的方案数。
该问题可分为两种情况:

- $m<=r$, 此时不管怎么走都满足条件, 贡献为 $\binom{n+m}{n}$。
- $m>r$, 记对称轴 $x=y$ 分别与 $x=r$, $y=m$ 相交于点 A,B, 考虑第一次穿过 AB 上的情况。对于 AB 上的某点 $(x,x), x\in[r,m)$, 无穿越到达该点的方案数为 $\binom{x}{x-r}-\binom{x}{x-r-1}$, 穿越该点到达终点的方案数为 $\binom{r+n-x}{m-x-1}$。

例题 Luogu P3336 [ZJOI2013] 话旧 [6]
容易发现题中 x 坐标相邻的点之间的函数图象方案是独立的, 排序后对每对相邻点计算最值和方案数。方案数受到函数最小值为 0 的限制。计算组合数时使用 CRT+Lucas。
此题源代码有误(也许是分析错了)。

```
0 #include <algorithm>
  #include <cstdio>
  int read() {
      int res = 0, c;
```

[5]Catalan number - Wikipedia https://en.wikipedia.org/wiki/Catalan_number
[6]https://www.luogu.org/problemnew/show/P3336

```
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1000005;
struct Point {
    int x, fx;
    bool operator<(const Point& rhs) const {
        return x < rhs.x;
    }
    bool operator==(const Point& rhs) const {
        return x == rhs.x;
    }
} P[size];
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
template <int mod>
struct Base {
    int fac[mod], inv[mod], invFac[mod];
    Base() {
        fac[0] = fac[1] = inv[1] = invFac[0] =
            invFac[1] = 1;
    }
    void pre(int x) {
        static int cx = 1;
        while(cx < x) {
            ++cx;
            fac[cx] = asInt64(fac[cx - 1]) * cx % mod;
            inv[cx] = asInt64(mod - mod / cx) *
                inv[mod % cx] % mod;
            invFac[cx] = asInt64(invFac[cx - 1]) *
                inv[cx] % mod;
        }
    }
    Int64 C(int n, int m) {
        pre(n);
        return asInt64(fac[n]) * invFac[m] % mod *
            invFac[n - m] % mod;
    }
    Int64 lucas(int n, int m) {
        Int64 res = 1;
        while(n && m && res) {
            int cn = n % mod, cm = m % mod;
            n /= mod, m /= mod;
            if(n < m)
```

```
                return 0;
            res = res * C(cn, cm) % mod;
        }
        return res;
    }
    Int64 getInv(Int64 a) {
        Int64 res = 1;
        int k = mod - 2;
        while(k) {
            if(k & 1)
                res = res * a % mod;
            k >>= 1, a = a * a % mod;
        }
        return res;
    }
    Int64 getFac(int n, int m, int k) {
        return lucas(n, m) * k * getInv(k);
    }
};
const int m1 = 7, m2 = 2848631, m3 = m1 * m2;
Base<m1> Ca;
Base<m2> Cb;
Int64 C(int n, int m) {
    if(n < m || m < 0)
        return 0;
    return (Ca.getFac(n, m, m2) +
            Cb.getFac(n, m, m1)) %
        m3;
}
Int64 count(int fl, int l, int r, int dx) {
    if(fl >= r)
        return C(dx, l);
    else {
        Int64 res = 0;
        for(int i = fl; i < r; ++i)
            res = (res +
                   (C(i, i - fl) - C(i, i - fl - 1)) *
                       C(fl + l - i, r - 1 - i)) %
                m3;
        return (C(dx, l) - res) % m3;
    }
}
void solve(Point pl, Point pr, int& maxf, int& cnt) {
    int dx = pr.x - pl.x, dy = pr.fx - pl.fx;
    int l = (dx + dy) >> 1, r = dx - l;
    maxf = std::max(maxf, pl.fx + l);
    cnt = cnt * (dy >= 0 ? count(pl.fx, l, r, dx) :
                           count(pr.fx, r, l, dx)) %
        m3;
}
```

```
int main() {
    int n = read();
    int k = read();
    for(int i = 0; i < k; ++i) {
        P[i].x = read();
        P[i].fx = read();
    }
    P[k].x = P[k].fx = P[k + 1].fx = 0;
    P[k + 1].x = n;
    k += 2;
    std::sort(P, P + k);
    k = std::unique(P, P + k) - P;
    int maxf = 0, cnt = 1;
    for(int i = 1; i < k; ++i)
        solve(P[i - 1], P[i], maxf, cnt);
    printf("%d %d\n", (cnt + m3) % m3, maxf);
    return 0;
}
```

6.3 Stirling 数

6.3.1 第一类 Stirling 数

令 $\begin{bmatrix} n \\ k \end{bmatrix}$ 为把 n 个点放到 k 个环内的方案数, 有

$$\begin{aligned} \begin{bmatrix} n \\ 0 \end{bmatrix} &= 0 \text{ for all } n \geq 1 \\ \begin{bmatrix} n \\ n \end{bmatrix} &= 1 \text{ for all } n \geq 0 \\ \begin{bmatrix} n \\ k \end{bmatrix} &= (n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix} \\ \sum_{i=0}^{n} \begin{bmatrix} n \\ i \end{bmatrix} &= n! \end{aligned}$$

证明递推式:

- 若将当前点丢给之前的环, 则可以选择 $n-1$ 个点在其右边, 因此贡献 $(n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix}$。
- 当前点自成一环, 贡献 $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$。

6.3.2 第二类 Stirling 数

令 $\left\{ {n \atop k} \right\}$ 为把 n 个点放到 k 个集合内的方案数, 有

$$\left\{ {n \atop 0} \right\} = 0 \text{ for all } n \geq 1 \tag{6.6}$$

$$\left\{ {n \atop n} \right\} = 1 \text{ for all } n \geq 0 \tag{6.7}$$

$$\left\{ {n \atop k} \right\} = k \left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\} \tag{6.8}$$

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n \tag{6.9}$$

证明等式 6.8:

- 若将当前点丢给之前的集合, 则可以选择 k 个集合, 因此贡献 $k \left[{n-1 \atop k} \right]$。
- 当前点自成一集合, 贡献 $\left\{ {n-1 \atop k-1} \right\}$。

证明 6.9: 考虑计算将 n 个点放入 k 个带编号集合且无空集的方案数:

$$\begin{aligned} N &= k^n + \sum_{i=1}^{k} (-1)^i \binom{k}{i} (k-i)^n \text{（根据定理 5.4）} \\ &= k^n + \sum_{i=0}^{k-1} (-1)^{k-i} \binom{k}{i} i^n \\ &= \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n \end{aligned}$$

由于要求的是无标号的方案数, 因此最终答案需要除以 $k!$。

一个常用的转化:

定理 6.2

$$n^k = \sum_{i=0}^{n} \left\{ {k \atop i} \right\} \binom{n}{i} i!$$

意义: 左边是将 k 个球放入 n 个盒子里的方案数, 右边枚举非空盒子数, 那么 i 个非空盒子的方案数即为划分非空子集的方案数 * 盒子排列数(盒子选取方案数 * 盒子标号方案数)。

6.4 贝尔数

贝尔数 B_n 是大小为 n 的集合的划分方案数。

贝尔数的前几项为: $1, 1, 2, 5, 15, 52, 203, 877, \cdots$ (参见 OEIS-A000110 https://oeis.org/A000110)

B_n 满足等式：

$$\begin{aligned} B_0 = B_1 &= 1 \\ B_{n+1} &= \sum_{i=0}^{n} \binom{n}{i} B_i \\ B_n &= \sum_{i=1}^{n} \left\{ {n \atop i} \right\} \end{aligned}$$

6.4.1 因子分解应用

若无平方因子的整数 $n = \sum_{i=1}^{k} p_i$, 则整数 n 可以分解为 B_k 种因子的乘积表达式。

6.4.2 贝尔数计算

6.4.2.1 DFS

若要得到不同方案的状态表示, 限制下阶段 DFS 的集合大小以避免重复计算。

6.4.2.2 贝尔三角形

与推杨辉三角形类似, 递推方法如下：

$$\begin{aligned} A[0][1] &= 1 \\ A[n][1] &= A[n-1][n] \\ A[n][m] &= A[n][m-1] + A[n-1][m-1] \\ B_n &= A[n][1] \end{aligned}$$

使用滚动数组优化空间。

6.4.2.3 生成函数法

利用 Bell 数的生成函数 e^{e^x-1}, 计算多项式 exp。

6.4.2.4 贝尔数模质数

定理 6.3 *Touchard's Congruence*

$$\begin{aligned} B_{p+n} &\equiv B_n + B_{n+1} \pmod{p} \\ B_{p^m+n} &\equiv mB_n + B_{n+1} \pmod{p} \end{aligned}$$

首先 $O(p^2)$ 使用贝尔三角形预处理出 p 以内的贝尔数模 p 的值。

若要求 B_n, 则对 n 进行 p 进制分解, 记 n_i 为 n 在权为 p^i 时的位。

以最低位作为下标, 从 $B_{0...p}$ 开始递推, 逐步加 p^i 递推直至次数等于 n_i。最后取数组第 n_0 项。

参考代码(PA2008 Cliquers Strike Back)：

```
#include <cstdio>
typedef long long Int64;
Int64 powm(Int64 a, int k, int mod) {
    a %= mod;
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
const int size = 7285, mod = 999999599, phi = mod - 1;
int dp[2][size], b[size];
int add(int a, int b) {
    a += b;
    return a >= phi ? a - phi : a;
}
void pre() {
    dp[0][1] = b[0] = 1;
    for(int i = 1, o = 0, c = 1; i <= 7283;
        ++i, o ^= 1, c ^= 1) {
        b[i] = dp[c][1] = dp[o][i];
        for(int j = 1; j <= i; ++j)
            dp[c][j + 1] = add(dp[c][j], dp[o][j]);
    }
}
int c[size], bit[70];
int solveImpl(Int64 n, int k) {
    int bsiz = 0;
    while(n) {
        bit[bsiz++] = n % k;
        n /= k;
    }
    for(int i = 0; i <= k; ++i)
        c[i] = b[i] % k;
    for(int i = 1; i < bsiz; ++i) {
        for(int j = 1; j <= bit[i]; ++j) {
            for(int x = 0; x < k; ++x)
                c[x] = (i * c[x] + c[x + 1]) % k;
            c[k] = (c[0] + c[1]) % k;
        }
    }
    return c[bit[0]];
}
const int fac[4] = { 2, 13, 5281, 7283 };
int solve(Int64 n) {
    Int64 res = 0;
    for(int i = 0; i < 4; ++i) {
        int k = fac[i];
```

```
        Int64 v = solveImpl(n, k);
        Int64 u = phi / k;
        res = (res + powm(u, k - 2, k) * u % phi * v) %
            phi;
    }
    return res;
}
int main() {
    Int64 n, m;
    scanf("%lld%lld", &n, &m);
    pre();
    printf("%lld\n", powm(m, solve(n), mod));
    return 0;
}
```

时间复杂度 $O(p^2 lg_p n)$。

该内容参考了 Claris 的博客[7]和 Wikipedia-EN[8]。**令人失望的是维基百科已全面被封。**

6.5 Lucas/ExLucas

6.5.1 Lucas 定理

定理 6.4 (Lucas's Theorem) 对于非负整数 n, m 以及素数 p,若

$$\begin{aligned} n &= \sum_{i=0}^{k} n_i p^i \\ m &= \sum_{i=0}^{k} m_i p^i \end{aligned}$$

则

$$\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod{p}$$

Nathan Fine 的证明:对于素数 p 与整数 n 满足 $1 \le n < p$, 有

引理 6.5

$$p \Big| \binom{p}{n} = \frac{p \cdot (p-1) \cdots (p-n+1)}{n \cdot (n-1) \cdots 1}$$

证明:注意到 p 是素数且与分母的每一个数互质, 不可被分母的因子约去, 所以最终值必有因子 p。那么可用普通生成函数表达为:

$$(1+x)^p \equiv 1 + x^p \pmod{p}$$

可归纳推广为

[7]BZOJ3501 : PA2008 Cliquers Strike Back https://www.cnblogs.com/clrs97/p/4714467.html

[8]Bell number - Wikipedia:Modular arithmetic https://en.wikipedia.org/wiki/Bell_number#Modular_arithmetic

推论 6.6

$$(1+x)^{p^i} \equiv 1 + x^{p^i} \pmod{p}, i \in \mathbb{N}$$

利用生成函数证明:

$$\begin{aligned}
\sum_{m=0}^{n} \binom{n}{m} x^m &= (1+x)^n \\
&= \prod_{i=0}^{k} ((1+x)^{p^i})^{n_i} \\
&\equiv \prod_{i=0}^{k} (1+x^{p^i})^{n_i} \text{（根据推论 6.6）} \\
&= \prod_{i=0}^{k} \left(\sum_{m_i=0}^{n_i} \binom{n_i}{m_i} x^{p_i m_i} \right) \\
&= \prod_{i=0}^{k} \left(\sum_{m_i=0}^{p-1} \binom{n_i}{m_i} x^{p_i m_i} \right) \\
&= \sum_{m=0}^{n} \left(\prod_{i=0}^{k} \binom{n_i}{m_i} \right) x^m \pmod{p}
\end{aligned}$$

以上内容参考了 Wikipedia-EN[9]

6.5.2 ExLucas

对于模数为合数的情况, 将 p 质因数分解, 即 $p = \sum_{i=1}^{k} p_i^{c_i}$。然后快速求出 $\binom{n}{m} \bmod p_i^{c_i}$ 的值, 最后使用 CRT 合并。

首先将求组合数转换为求阶乘, 但要从 $n!$ 提出 p 的倍数并做特殊处理, 即

$$n! = \prod_{i=1}^{n} i^{[p \nmid i]} \cdot p^{[\frac{n}{p}]} \cdot \prod_{i=1}^{[\frac{n}{p}]} i$$

- 第一部分:前 $[\frac{n}{p_i^{c_i}}]$ 块的答案相等, 计算整块后快速幂, 末尾不完整的块暴力计算。
- 第二部分:由于组合数中分子分母都有因子 p, 单独拆出来算。
  统计次数代码:

```
int cnt = 0;
for(int i=n/p;i;i/=p)
    cnt+=i;
```

- 第三部分:作为一个子问题递归解决。

对于单个素数幂计算复杂度 $O(p_i^{c_i})$。

以上内容参考了 Candy? 的博客[10]。

[9]Lucas's theorem - Wikipedia https://en.wikipedia.org/wiki/Lucas%27s_theorem
[10][Lucas 定理]【学习笔记】- Candy? https://www.cnblogs.com/candy99/p/6637629.html

6.6 康托展开

康托展开可用于将一个排列映射到一个自然数(在所有排列中的字典序排名, 从 0 开始)。

排列 P 的康托展开为 $\sum_{i=2}^{n}(i-1)!a_i$, 其中 a_i 为位置 i 后小于 P_i 的数的个数, 即 $\sum_{j=1}^{i-1}[P_i > P_j]$。

相反已知字典序排名可以将其映射到一个排列, 这就是逆康托展开。首先根据排名值计算出 a 数组, 数组 a 是唯一确定的, 因为 $a_i < i$ 而前一项的系数 $i! > (i-1)!a_i$, 从前到后取模确定。计算出数组 a 后, 维护一个支持删除和询问第 k 大的数据结构, 从前到后确定每一位数。

6.7 其它公式与定理

6.7.1 常用公式

定理 6.7

$$\sum_{i=0}^{n}\binom{i}{k}=\binom{n+1}{k+1}$$

- 证明:将等式加上 $\binom{0}{k+1}=0$, 左边不断合并即为右式。
- 证明:将右式不断展开即为左式。
- 理解:在 $n+1$ 个中选 $k+1$ 个点, 第 $k+1$ 个点为哨兵, 剩下 k 个在它之前的元素中选择。

定理 6.8

$$\sum_{i=0}^{n}\binom{n}{i}i=n2^{n-1}$$

定理 6.9

$$\int_0^1\binom{n}{k}x^k(1-x)^{n-k}dx=\frac{1}{n+1}$$

定理 6.10

$$(-1+1)^n=\sum_{k=0}^{n}(-1)^k\binom{n}{k}=[n=0]$$

更一般地, 可以得到

定理 6.11

$$\sum_{k=0}^{m}(-1)^k\binom{n}{k}=\binom{m-n}{m}$$

证明:

引理 6.12 (上指标反转)

$$\binom{n}{m} = (-1)^m \binom{m-n-1}{m}$$

将组合数的分子取反可证。由于 $(-1)^i = (-1)^{-i}$, 该式也可表述为

推论 6.13

$$(-1)^m \binom{n}{m} = \binom{m-n-1}{m}$$

接下来证明定理:

$$\begin{aligned}\sum_{k=0}^{m} (-1)^k \binom{n}{k} &= \sum_{k=0}^{m} \binom{k-n-1}{k} \\ &= \binom{m-n}{m} \text{(展开后不断合并最左边两项)}\end{aligned}$$

6.7.2 Raney 引理

引理 6.14 (Raney Lemma) 设整数序列为 $A_1, A_2, \cdots, A_n$, 记前缀和 $S_k = \sum_{i=1}^{k} A_k$, 满足 $S_n = 1$。则在该序列的循环表示中, 有且只有一个序列满足所有前缀和均大于 0。任何一种循环表示都和自身不同。

证明留坑待补。

例题 SP9507 MARIO2 - Mario and Mushrooms
可将循环同构的序列归类, 根据 Raney 引理, 每类序列只有 1 个序列满足题意。

```
#include <cstdio>
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 1; i <= n; ++i) {
        int m, k;
        scanf("%d%d", &m, &k);
        printf("Case #%d: %.8lf\n", i,
               1.0 / (m * k + 1 + k));
    }
    return 0;
}
```

上述内容参考了兔子大天使的博客[11]。

[11]Raney 引理 https://blog.csdn.net/duxingstar/article/details/6406022

Chapter 7

多项式

7.1 快速傅里叶变换 FFT 215
7.1.1 FFT 原理 215
7.1.2 迭代 FFT 实现 217
7.1.3 实序列 DFT 219
7.1.4 MTT 之拆系数 FFT 220
7.2 快速数论变换 NTT 220
7.2.1 NTT 原理 220
7.2.2 NTT 实现 220
7.2.3 NTT 常见模数 220
7.2.4 MTT 之三模数 NTT 221
7.3 快速沃尔什变换 FWT 221
7.3.1 FWT 原理 221
7.3.2 nand,nor,nxor 222
7.3.3 FWT 实现 222
7.4 快速莫比乌斯变换/反演 224
7.4.1 子集卷积 226
7.5 多项式高级算法 226
7.5.1 牛顿迭代法 226
7.5.2 多项式开方 227
7.5.3 多项式求逆 227
7.5.4 多项式取模 227
7.5.5 多项式求导与积分 228
7.5.6 多项式 ln 228
7.5.7 多项式 exp 228
7.5.8 多项式快速幂 228
7.5.9 多项式三角函数 228
7.5.10 进制转换 235
7.5.11 多项式多点求值 235
7.5.12 多项式多点插值 240
7.5.13 组合数取模 241
7.5.14 CDQ 分治 FFT 241
7.5.15 多项式乘积(普通分治 FFT) 241

7.5.16 二元多项式卷积 242
7.5.17 循环卷积 242
7.5.18 CZT 245
7.6 多项式算法封装 246
7.7 计算形式幂级数的牛顿迭代法的常数优化 248
7.7.1 卷积 248
7.7.2 求逆 248
7.7.3 平方根 249
7.7.4 指数 249
7.7.5 优化方法总结与实现细节 251
7.7.6 实验结果 251
7.8 本章注记 252

7.1 快速傅里叶变换 FFT

7.1.1 FFT 原理

FFT 求多项式卷积的过程为:$\Theta(n\lg n)$ 求值 $\Rightarrow \Theta(n)$ 点值乘法 $\Rightarrow \Theta(n\lg n)$ 插值。$\Theta(n\lg n)$ 求值/插值的复杂度是在单位复数根上计算得到的。

7.1.1.1 单位复数根

定义n **次单位复数根**是满足 $\omega^n = 1$ 的复数 ω, 恰好有 n 个, 即 $\omega_n^k = e^{2\pi ik/n}, k = 0, 1, \cdots, n-1$。

定义**主** n **次单位根**$\omega_n = e^{2\pi i/n}$。

下面是关于 n 次单位复数根的性质:

引理 7.1 (消去引理) 对于任意整数 $n \geq 0, k \geq 0, d > 0$,

$$\omega_{dn}^{dk} = \omega_n^k$$

证明:

$$\omega_{dn}^{dk} = e^{2\pi idk/dn} = e^{2\pi ik/n} = \omega_n^k$$

推论 7.2 对于任意偶数 $n > 0$,有

$$\omega_n^{n/2} = \omega_2 = -1$$

引理 7.3 (折半引理) 对于偶数 $n > 0$,n 个 n 次单位复数根的平方组成的集合为 $n/2$ 个 $n/2$ 次单位复数根的集合。

证明:根据引理 7.1可得 $(\omega_n^k)^2 = (\omega_n^{k+n/2})^2 = \omega_{n/2}^k$, 每个 $n/2$ 次单位复数根恰好被得到 2 次。

引理 7.4 (求和引理) 对于任意整数 $n \geq 1$ 与不能被 n 整除的非负整数 k,有

$$\sum_{i=0}^{n-1} (w_n^k)^i = 0$$

证明:

$$\sum_{i=0}^{n-1} (w_n^k)^i = \frac{(w_n^k)^n - 1}{w_n^k - 1} = 0$$

n 不整除 k 保证了分母不为 0。

7.1.1.2 DFT

对于次数界为 n 的多项式

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

其 DFT 为

$$DFT_n(a) = (y_0, y_1, \cdots, y_{n-1}) = (A(\omega_n^0), A(\omega_n^1), \cdots, A(\omega_n^{n-1}))$$

7.1.1.3 FFT

FFT 采用分治策略,假设 n 是 2 的幂(不足补 0),其步骤如下:

1. 若次数界为 1,则返回 a_0。
2. 定义新的次数界为 $n/2$ 多项式

$$\begin{aligned} A^{[0]}(x) &= a_0 + a_2 x + \cdots + a_{n-2} x^{n/2-1} \\ A^{[1]}(x) &= a_1 + a_3 x + \cdots + a_{n-1} x^{n/2-1} \end{aligned}$$

   递归计算其在点 $(\omega_n^0)^2, (\omega_n^1)^2, \cdots, (\omega_n^{n-1})^2$ 的值(实际上递归只求了前一半)。
3. 该多项式满足等式

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{7.1}$$

   可利用递归计算的值合并。对于 $k = 0, 1, \cdots, n/2 - 1$,

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ y_{k+n/2} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \end{aligned}$$

   正确性证明:

$$\begin{aligned} y_k &= y_k^{[0]} + \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_{n/2}^k) + \omega_n^k A^{[1]}(\omega_{n/2}^k) \\ &= A^{[0]}(\omega_n^{2k}) + \omega_n^k A^{[1]}(\omega_n^{2k}) \text{ (根据引理 7.1)} \\ &= A(\omega_n^k) \text{ (根据式 7.1)} \\ y_{k+n/2} &= y_k^{[0]} - \omega_n^k y_k^{[1]} \\ &= A^{[0]}(\omega_{n/2}^k) + \omega_n^{k+n/2} A^{[1]}(\omega_{n/2}^k) \text{ (根据推论 7.2)} \\ &= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+n/2} A^{[1]}(\omega_n^{2k+n}) \text{ (根据引理 7.1与 } \omega_n^n = 1) \\ &= A(\omega_n^{k+n/2}) \text{ (根据式 7.1)} \end{aligned}$$

7.1.1.4 逆 DFT

定理 7.5 对于范德蒙德矩阵 V_n 满足 $v_{ij} = \omega_n^{ij}$,其逆矩阵 V_n^{-1} 满足 $v_{ij}^{-1} = \omega_n^{-ij}/n$(矩阵下标从 *0* 开始)。

证明:

$$\begin{aligned}[V_nV_n^{-1}]_{ij} &= \sum_{k=0}^{n-1}\omega_n^{ik}\omega_n^{-kj}/n \\ &= \sum_{k=0}^{n-1}\omega_n^{k(i-j)}/n \\ &= [i=j]\text{（根据引理 7.4）} \\ \Rightarrow\ & V_nV_n^{-1}=I_n\end{aligned}$$

所以逆 DFT 的矩阵就是 DFT 矩阵的 $\frac{1}{n}$, 用 ω_n^{-1} 代替 ω_n, 对多项式的 DFT 再做一次 DFT 后除以 n 就能得到原多项式。

以上内容来自算法导论 [4] 第 30 章多项式与快速傅里叶变换。

7.1.2 迭代 FFT 实现

7.1.2.1 单位复数根预处理

一般可以确定 FFT 所需最大规模, 因此可以在 FFT 前预处理。

```
typedef double FT;
typedef std::complex<FT> Complex;
int tot;
Complex root[size],invR[size];
void init(int n) {
    tot=n;
    FT base=2.0*acos(-1.0)/n;
    for(int i=0;i<n;++i) {
        root[i]=Complex(cos(base*i),sin(base*i));
        invR[i]=conj(root[i]);
    }
}
```

根据引理 7.1, $root[tot/n*k]=w_n^k$。

7.1.2.2 离线位逆序置换

为了做迭代 FFT, 我们需要置换原多项式的系数, 第 i 个系数被换到基于 2^k 的 i 的位逆序上。若 FFT 的规模不变, 比如求多项式乘法之类的简单问题, 可预处理位逆序置换数组后, 按照数组下标置换(规定 $i<rev[i]$ 保证只置换 1 次)。

```
int rev[size];
void initRev(int k) {
    rev[0]=0;
    int end=1<<k;
    for(int i=1;i<end;++i)
        rev[i]=rev[i>>1]>>1|(i&1)<<(k-1);
}
```

7.1.2.3 在线位逆序置换

在分治 FFT 时, 需要不同规模的位逆序置换, 预处理置换数组不太适用。这里介绍的是时间复杂度 $O(n)$, 空间复杂度 $O(1)$ 的在线算法。

```
void rev(int* A,int n) {
    for(int i=0,j=0;i<n;++i) {
        if(i>j)std::swap(A[i],A[j]);
        int l=n>>1;
        while((j^=l)<l)l>>=1;//key
    }
}
```

在每轮循环开始时, i 与 j 互为位逆序。代码中 key 部分是整个算法的关键, i 的递增就是正向二进制加法, j 就要进行反向二进制加法, 即从高到低找到第 1 个 0, 从高到低将这一段取反。

时间复杂度证明: 设 $n = 2^k$, 按 $while$ 迭代次数分类, 有

$$\begin{aligned} T(n) &= \sum_{i=1}^{k} i \cdot 2^{k-i} \\ &= n \sum_{i=1}^{k} i \cdot 2^{-i} \\ &= n \sum_{i=1}^{k} \sum_{j=i}^{k} 2^{-j} \\ &< n \sum_{i=1}^{k} 2^{1-i} \\ &< 2n \Rightarrow T(n) = O(n) \end{aligned}$$

7.1.2.4 迭代 FFT

考虑每一层递归树, 发现最底层的递归树是按位逆序顺序排的(每一次递归相当于一次 01 划分)。若系数数组按照这种顺序存储, 则可以很快地找到每一层对应的位置。

迭代 FFT 的计算顺序是自底向上的, 代码如下:

```
void FFT(int n, Complex *A, Complex *w) {
    for (int i = 0, j = 0; i < n; ++i) {
        if (i > j) std::swap(A[i], A[j]);
        int l = n >> 1;
        while ((j ^= l) < l) l >>= 1;
    }
    for (int i = 2; i <= n; i <<= 1) {
        int m = i >> 1, fac = tot / i;
        for (int j = 0; j < n; j += i)
            for (int k = 0; k < m; ++k) {
                Complex t = A[j + k + m] * w[k * fac];
                A[j + k + m] = A[j + k] - t;
                A[j + k] += t;
            }
```

```
    }
}
```

首先做位逆序置换, 然后从倒数第二层开始向上递推, i 表示当前层每一块的长度, m 是下一层每一块的长度, fac 是单位复数根缩放系数, j 是块编号, 内层循环与递归 FFT 合并答案的方法相同, 注意蝴蝶操作的求值顺序。

以上内容参考了 Miskcoo 的博客[1]。

7.1.3 实序列 DFT

对于多项式乘法这类问题, 常规方法需要对每个序列做 1 次 DFT。但由于我们只在实数域上做 FFT, 可以考虑将两个实序列合并为一个复序列, 直接使用 1 次 DFT 以及一点后处理得到两个实序列的 DFT。

记大小为 N 的序列 a 的 DFT 为 A, 考虑序列 A 的共轭为 $\overline{A}$:

$$\overline{A(n)} = \sum_{k=0}^{N-1} \overline{a(k)} \cdot \omega_N^{-nk} = \sum_{k=0}^{N-1} \overline{a(k)} \cdot \omega_N^{K(N-n)} n = 0, 1, \cdots, N-1$$

对于实序列 a, 有 $a(n) = \overline{a(n)}$, 由此可得 $\overline{A(n)} = A(N-n)$, 即实序列的 DFT 共轭对称。

令 $z(i) = x(i) + i \cdot y(i)$, 则 $Z(n) = X(n) + i \cdot Y(n)$, 记 $Z(n)$ 的实部与虚部分别为 $X(n)$ 与 $Y(n)$, 有

$$\begin{aligned} \overline{Z(N-n)} &= \overline{X(N-n) + i \cdot Y(N-n)} \\ &= \overline{A(N-n) + i \cdot B(N-n)} \\ \Rightarrow X(n) - i \cdot Y(n) &= A(N-n) - i \cdot B(N-n) \end{aligned}$$

联立解得

$$\begin{aligned} 2X(n) &= (A(n) + A(N-n)) + i \cdot (B(n) - B(N-n)) \\ 2Y(n) &= (B(n) + B(N-n)) - i \cdot (A(n) - A(N-n)) \end{aligned}$$

对复序列 $z(n)$ 的 DFT 后处理一下就可以得到 $X(n)$ 与 $Y(n)$(注意要对 $n = 0$ 进行特别处理)。

代码如下:

```
0 FFT(p,A,root);
  X[0] = A[0].real();
  Y[0] = A[0].imag();
  for (int i = 1; i < p; ++i) {
      FT x1 = A[i].real(), y1 = A[i].imag();
      FT x2 = A[p - i].real(), y2 = A[p - i].imag();
      X[i] = Complex((x1 + x2) * 0.5, (y1 - y2) * 0.5);
      Y[i] = Complex((y1 + y2) * 0.5, (x2 - x1) * 0.5);
  }
```

以上内容参考了 Miskcoo 的博客[2]。

注意此法的精度不如常规方法,在 CF553E Kyoya and Train 一题中尤其明显。一般仅用于 MTT。

[1]从多项式乘法到快速傅里叶变换–Miskcoo's Space http://blog.miskcoo.com/2015/04/polynomial-multiplication-and-fast-fourier-transform

[2]实序列离散傅里叶变换的快速算法–Miskcoo's Space http://blog.miskcoo.com/2018/01/real-dft

7.1.4 MTT 之拆系数 FFT

MTT 主要用来解决模数不满足 NTT 条件(模数无原根或者欧拉函数值中 2 的幂次比数据范围小)的卷积问题, 可以使用拆系数 FFT 的实数运算来绕开限制。

设模数为 M, 解决方法是找一个 $k \geq \sqrt{M}$, 将序列 $a(n)$ 拆为 2 个新序列 $a_0(n) = a(n)/k, a_1(n) = a(n)\%k$, 两两相乘后按照系数合并。共 4 次 DFT, 4 次 IDFT。

优化:

- 注意到这 4 个多项式均为实序列, 所以可按照 7.1.3节中所述方法仅使用 2 次 DFT。
- 合并时有两个多项式的系数均为 k, 可以加在一起 IDFT, 仅使用 3 次 IDFT。

7.2 快速数论变换 NTT

7.2.1 NTT 原理

NTT 的原理与 FFT 类似, 即找到单位根 x 满足 $x^n \equiv 1 \pmod p$。NTT 模数 p 需满足 p 为素数且 $p = r \cdot 2^k + 1$。

根据定理 4.7可知若模数 p 为素数则有 $x^{p-1} \equiv 1 \pmod p$, 所以当 $n|(p-1)$ 时才能进行 NTT。

根据 4.11节所述, p 必有原根, 设 p 的原根为 g, 则 $g^{\frac{p-1}{n}}$ 就是**主 n 次单位根**, n 个单位根即为 $w_n^k = g^{k \cdot \frac{p-1}{n}}$。

其余部分与 FFT 相同。

7.2.2 NTT 实现

NTT 仅预处理单位复数根部分不同, 以模 998244353 为例:

```
int tot, root[size], invR[size];
void init(int n) {
    const int g = 3;
    tot = n;
    Int64 base = powm(g, (mod - 1) / n);
    Int64 invBase = powm(base, mod - 2);
    root[0] = invR[0] = 1;
    for (int i = 1; i < n; ++i)
        root[i] = root[i - 1] * base % mod;
    for (int i = 1; i < n; ++i)
        invR[i] = invR[i - 1] * invBase % mod;
}
```

7.2.3 NTT 常见模数

- $469762049 = 7 * 2^{26} + 1$。
- $998244353 = 119 * 2^{23} + 1$。
- $1004535809 = 479 * 2^{21} + 1$, 加起来不爆 int。
- $2281701377 = 17 * 2^{27} + 1$, 平方恰好不爆 long long。

它们的原根均为 3。

7.2.4 MTT 之三模数 NTT

选取 3 个模数, 比如 {469762049, 998244353,1004535809}, 要求它们的乘积大于卷积过程中最大的数, 分别以这三个数为模数求 NTT, 最后使用 CRT 求解同余方程组。

但是使用 CRT 求解会爆 long long, 因此先合并前两项, 得到

$$\begin{aligned} x &\equiv n_1 \pmod{p_1} \\ x &\equiv n_2 \pmod{p_2} \end{aligned}$$

设 $x = k_1p_1 + n_1 = k_2p_2 + n_2$, 由于 $k_1 < p_2$, 我们可以求解 $k_1p_1 \equiv n_2 - n_1 \pmod{p_2}$ 得到 k_1, 带入原式求出 $x \bmod p$ 的值。

该方法来自 AntiLeaf[3] 的博客。

7.3 快速沃尔什变换 FWT

7.3.1 FWT 原理

FWT 主要用来求下列卷积:

$$z_n = \sum_{i \oplus j = n} a_i b_j$$

其中 $\oplus$ 为二元位运算符。

其原理与 FFT 相同, 关键在于蝴蝶操作。

序列可表示为 $A = (A_0, A_1)$, 01 表示当前处理的位, 只需找到对于 $C = A \otimes B$, 有 $FWT(C) = FWT(A) * FWT(B)$, 由 $UFWT(FWT(A)) = A$ 可推出其逆变换。

现场推导时可以仅考虑由倒数第二层合并至倒数第三层的情况, 其余自底向上可以归纳证明。

7.3.1.1 and

由 $C = (A_0 * B_0 + A_0 * B_1 + A_1 * B_0, A_1 * B_1)$ 可构造出

$$\begin{aligned} FWT_{and}(A) &= (FWT_{and}(A_0) + FWT_{and}(A_1), FWT_{and}(A_1)) \\ UFWT_{and}(A) &= (UFWT_{and}(A_0) - UFWT_{and}(A_1), UFWT_{and}(A_1)) \end{aligned}$$

7.3.1.2 or

$$\begin{aligned} FWT_{or}(A) &= (FWT_{or}(A_0), FWT_{or}(A_1) + FWT_{or}(A_0)) \\ UFWT_{or}(A) &= (UFWT_{or}(A_0), UFWT_{or}(A_1) - UFWT_{or}(A_0)) \end{aligned}$$

7.3.1.3 xor!!!

$$\begin{aligned} FWT_{xor}(A) &= (FWT_{xor}(A_0) + FWT_{xor}(A_1), FWT_{xor}(A_0) - FWT_{xor}(A_1)) \\ UFWT_{xor}(A) &= \left(\frac{UFWT_{xor}(A_0) + UFWT_{xor}(A_1)}{2}, \frac{UFWT_{xor}(A_0) - UFWT_{xor}(A_1)}{2}\right) \end{aligned}$$

[3]COGS2294 释迦 - AntiLeaf http://www.cnblogs.com/hzoier/p/6441967.html

7.3.2 nand,nor,nxor

求出 $\oplus$ 为 and,or,xor 时的 FWT 后按取反交换。
以上内容参考了 picks 的博客 [4]。

7.3.3 FWT 实现

FWT

```
// P4717
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
typedef long long Int64;
typedef void (*FwtFunc)(int*, int);
#define asInt64(x) static_cast<Int64>(x)
const int size = 1 << 18, mod = 998244353;
int add(int a, int b) {
    a += b;
    return a >= mod ? a - mod : a;
}
int sub(int a, int b) {
    a -= b;
    return a < 0 ? a + mod : a;
}
const Int64 inv2 = 499122177;
#define FWTBEGIN(func)                          \
    void func(int* A, int n) {                  \
        for(int i = 2; i <= n; i <<= 1) {       \
            int m = i >> 1;                     \
            for(int j = 0; j < n; j += i) {     \
                for(int k = 0; k < m; ++k) {

#define FWTEND \
    }          \
    }          \
    }          \
    }
```

[4] Fast Walsh-Hadamard Transform ≪ Picks's Blog
http://picks.logdown.com/posts/179290-fast-walsh-hadamard-transform

```
FWTBEGIN(FWTOr)
A[j + k + m] = add(A[j + k], A[j + k + m]);
FWTEND
FWTBEGIN(UFWTOr)
A[j + k + m] = sub(A[j + k + m], A[j + k]);
FWTEND
FWTBEGIN(FWTAnd)
A[j + k] = add(A[j + k], A[j + k + m]);
FWTEND
FWTBEGIN(UFWTAnd)
A[j + k] = sub(A[j + k], A[j + k + m]);
FWTEND

FWTBEGIN(FWTXor)
int t = A[j + k + m];
A[j + k + m] = sub(A[j + k], t);
A[j + k] = add(A[j + k], t);
FWTEND
FWTBEGIN(UFWTXor)
int t = A[j + k + m];
A[j + k + m] = sub(A[j + k], t) * inv2 % mod;
A[j + k] = add(A[j + k], t) * inv2 % mod;
FWTEND

int A[size], B[size], C[size], D[size];
void foo(int n, FwtFunc fwt, FwtFunc ufwt) {
    memcpy(C, A, sizeof(int) * n);
    memcpy(D, B, sizeof(int) * n);
    fwt(C, n);
    fwt(D, n);
    for(int i = 0; i < n; ++i)
        C[i] = asInt64(C[i]) * D[i] % mod;
    ufwt(C, n);
    for(int i = 0; i < n; ++i)
        printf("%d ", C[i]);
    puts("");
}
int main() {
    int n = 1 << read();
    for(int i = 0; i < n; ++i)
        A[i] = read();
    for(int i = 0; i < n; ++i)
        B[i] = read();
    foo(n, FWTOr, UFWTOr);
    foo(n, FWTAnd, UFWTAnd);
    foo(n, FWTXor, UFWTXor);
    return 0;
}
```

7.4 快速莫比乌斯变换/反演

快速莫比乌斯变换用于计算集合并卷积。快速莫比乌斯反演其实就是其逆变换。实际上这就是 FWT 的非递归形式。

给定数组 f, g, 求 $f * g$ 的集合并卷积 $h[k] = \sum_{i|j==k} f[i]g[j]$, 数组规模为 2^n。

记数组 F 为 f 的莫比乌斯变换, 满足 $F[i] = \sum_{i\&j==j} f[j]$ (实际上就是子集和)。同理计算数组 g 的莫比乌斯变换 G。令 $H[i] = F[i] * G[i]$, 会发现 $H[i] = \sum_{i\&j==j} h[j]$。

接下来考虑如何从 f 快速推出 F: 枚举每一个比特位, 对每一比特位做前缀和, 这样就可以保证得到正确结果。时间复杂度 $O(2^n n)$。

```
0 int end = 1 << n;
  for(int i = 1; i < end; i <<= 1)
      for(int j = 0; j < end; ++j)
          if(j & i)
              A[j] += A[j ^ i];
```

注意到所有的 j 都是 i 的父集, 且 A 值改变不传递, 因此可以枚举 i 的父集去掉判断。不过实际优化效果并达不到 2 倍, 可能性能瓶颈在于存取。

```
0 int end = 1 << n;
  for(int i = 1; i < end; i <<= 1)
      for(int j = i; j < end; j = (j + 1) | i)
          A[j] += A[j ^ i];
```

Update: 似乎 FWT 比 FMT 更快些, 可能它更 Cache-Friendly 吧。

逆变换只要将其倒着做就可以了 (从高到低枚举与从低到高枚举等价, A 值改变的影响不会传递, 所以不必控制枚举顺序)。

```
0 int end = 1 << n;
  for(int i = 1; i < end; i <<= 1)
      for(int j = 0; j < end; ++j)
          if(j & i)
              A[j] -= A[j ^ i];
```

用容斥可得 $h[i] = \sum_{i\&j==j} (-1)^{|I|-|J|} H[j]$, 其中 $|I|$ 表示 i 对应的集合 I 的大小。

证明: 该式的 $H[j]$ 子集中含有 $h[i]$ 当且仅当 $i == j$, 贡献为 $h[i]$。考虑其余 $h[k]$ 对 $H[j]$ 的贡献, 以及它们在式中最终对 $h[i]$ 的贡献, 设 $d = |I| - |K|$, $h[k]$ 对 $h[i]$ 的贡献为 $\sum_{i=0}^{2^d-1} (-1)^{d-|i|}$。由于选择奇数位与选择偶数位的方案数相等, 贡献为 0。

注意事项 单次 FMT 的实际意义很重要, 经常用于辅助容斥预处理, 即使对集合幂级数的操作无定义。

例题 Luogu P3175 [HAOI2015] 按位或[5]

[5]【P3175】[HAOI2015] 按位或 - 洛谷 https://www.luogu.org/problemnew/show/P3175

记 a_i 为走 i 步到达目标状态的概率, 答案为 $\sum_{k=1}^{\infty} k(a_k - a_{k-1}) = -\sum_{k=0}^{\infty} a_k$。令概率数组为集合幂级数 f, 定义乘法运算为集合幂卷积, 可知答案为 f 的无限递减几何级数第 $2^n - 1$ 项的倒数, 若其收敛则其答案为 $\left(\frac{1}{f-1}\right)_{2^n-1}$。计算 f 的莫比乌斯变换 F, 那么函数 $T(f)$ 对应序列 $T(F_i)$。记 $G_i = \frac{1}{F_i - 1}$, 最后使用容斥计算出 g_{2^n-1}, 不必做反演。

若有解则 $F_{2^n-1} = 1$, 此时对应的 G_{2^n-1} 应该为 0, 所以容斥时不考虑目标状态项。

代码:

```
// P3175
#include <cctype>
#include <cstdio>
#include <cstdlib>
typedef double FT;
const FT eps = 1.0 - 1e-8;
FT read() {
    char str[30];
    int p = 0, c;
    do
        c = getchar();
    while(!isgraph(c));
    while(isgraph(c)) {
        str[p++] = c;
        c = getchar();
    }
    str[p] = '\0';
    return strtod(str, 0);
}
const int size = 1 << 20;
FT P[size], A[size];
int main() {
    int n;
    scanf("%d", &n);
    int end = 1 << n;
    for(int i = 0; i < end; ++i)
        P[i] = read();
    A[0] = (n & 1 ? -1.0 : 1.0);
    for(int i = 1; i < end; ++i)
        A[i] = -A[i - (i & -i)];
    for(int i = 1; i < end; i <<= 1)
        for(int j = 0; j < end; ++j)
            if(j & i)
                P[j] += P[j ^ i];
    bool flag = true;
    int e2 = end - 1;
    for(int i = 0; i < e2; ++i)
        if(P[i] >= eps) {
            flag = false;
            break;
        }
```

```
    if(flag) {
        FT ans = 0.0;
        for(int i = 0; i < e2; ++i)
            ans += A[i] / (P[i] - 1.0);
        printf("%.10lf\n", ans);
    } else
        puts("INF");
    return 0;
}
```

上述内容参考了 liu_runda 的博客[6]。

7.4.1 子集卷积

有时按照 1 的个数将原数组分解也是一个比较好的思路。

例题:LOJ161 子集卷积

分析后可得到要求的是 $h[k] = \sum_{i\&j=0 \wedge i|j=k} f[i]g[j]$。考虑除去约束 $i\&j = 0$, 由于我们能够做约束 $i|j = k$ 的卷积, 在满足这个约束的情况下, 约束 $i\&j = 0$ 可以等价为 $bitcount(i) + bitcount(j) = bitcount(k)$。那么将原数组按照 1 的个数分解, 然后对每类 $bitcount$ 卷积, 最后只要输出 $bitcount(k)$ 的 k 的值。

7.5 多项式高级算法

警告:慎卡常导致编码/调试困难。

Update:多项式算法模块化封装已完成, 参见 7.6节与 7.7节的内容。

7.5.1 牛顿迭代法

已知函数 $G(z)$, 求函数 $F(z) \bmod z^n$ 满足 $G(F(z)) \equiv 0 \pmod{z^n}$。

当 $n = 1$ 时, 多项式退化为常数, 直接求解。

记 $h = \lfloor \frac{n+1}{2} \rfloor$, 若已知 $G(F_0(z)) \equiv 0 \pmod{z^h}$, 尝试计算 $G(F(z))$ 在 $F_0(z)$ 处的泰勒展开:

$$G(F(z)) = \sum_{i=0}^{\infty} \frac{G^{(i)}(F_0(z))}{i!} \cdot (F(z) - F_0(z))^i$$

可以发现 $F(z)$ 与 $F_0(z)$ 的后 h 项均相等, 所以两多项式之差的 $n \geq 2$ 次方多项式的最小非 0 项次数 $\geq n$, 模 z^n 意义下无贡献, 因此仅前两项展开有效, 即

$$G(F(z)) \equiv G(F_0(z)) + G'(F_0(z))(F(z) - F_0(z)) \pmod{z^n}$$

结合 $G(F(z)) \equiv 0 \pmod{z^n}$ 可得到新的 $F(z)$:

$$F(z) \equiv F_0(z) - \frac{G(F_0(z))}{G'(F_0(z))} \pmod{z^n}$$

这就是牛顿迭代法。

[6] bzoj4036[HAOI2015]set 按位或 https://www.cnblogs.com/liu-runda/p/6443577.html 不会 FWT 的选手计算集合并卷积的方法 http://liu-runda.blog.uoj.ac/blog/2360

7.5.1.1 注意事项

- 使用 FFT 加速卷积后及时取模, 即把不需要的位置 0。否则循环卷积性质会干扰低次项。当然也可以直接将规模对齐到 2 的幂次, 甚至可以利用循环卷积优化, 最后一次再取模。
- 卷积时使用 > 2 倍模次数的 2 的幂作为卷积规模, 因为乘法操作至少需要这么多项才足够确定多项式系数。
- 若要求 $\bmod z^n$ 意义下的结果, 求导/积分这类导致多项式次数变化的操作, 显然仅把次数设为 n 会导致信息丢失, 而且讨论每种操作的最高次数也很麻烦。不妨全部求 $\bmod z^{n+c}$ 意义下的结果, c 为足够大的小常数, 最后直接截断输出。

7.5.2 多项式开方

对于给定的 $A(z)$, 求 $F(z) \pmod{z^n}$, 使得 $F^2(z) \equiv A(z) \pmod{z^n}$。
构造方程 $F^2(z) - A(z) \equiv 0 \pmod{z^n}$, 同理可得

$$\begin{aligned} F(z) &\equiv F_0(z) - \frac{F_0(z)^2 - A(z)}{2F_0(z)} \pmod{z^n} \\ &\equiv \frac{F_0(z)^2 + A(z)}{2F_0(z)} \pmod{z^n} \end{aligned}$$

注意当 $t = 0$ 时可能需要用二次剩余在模意义下开根。

7.5.3 多项式求逆

多项式求逆是基于牛顿迭代法的多项式算法的基本工具。
对于给定的 $A(z)$, 求 $F(z) \pmod{z^n}$, 使得 $F(z) \cdot A(z) \equiv 1 \pmod{z^n}$。
构造方程 $F(z) \cdot A(z) - 1 \equiv 0 \pmod{z^n}$, 同理可得

$$\begin{aligned} F(z) &\equiv F_0(z) - \frac{F_0(z)A(z) - 1}{A(z)} \pmod{z^n} \\ &\equiv F_0(z) - (F_0(z)A(z) - 1)F(z) \pmod{z^n} \\ &\equiv F_0(z) - (F_0(z)A(z) - 1)F_0(z) \pmod{z^n} \ (F_0(z)A(z) - 1 \equiv 0 \pmod{z^h}) \\ &\equiv 2F_0(z) - F_0^2(z)A(z) \pmod{z^n} \end{aligned}$$

7.5.4 多项式取模

给定一个 n 次多项式 $A(z)$ 与 m 次多项式 $B(z)(m \le n)$, 求多项式 $D(z), R(z)$ 满足

$$A(z) = D(z)B(z) + R(z), deg(D) \le n - m, deg(R) < m$$

记 n 次多项式 $A(z)$ 的系数翻转 $A^R(z) = z^n A(\frac{1}{z})$。

令 $deg(D) = n - m, deg(R) = m - 1$, 不足高位补 0。将原方程的 z 全部换成 $\frac{1}{z}$, 然后乘上 z^n, 有

$$\begin{aligned} z^n A(\frac{1}{z}) &= z^n D(\frac{1}{z})B(\frac{1}{z}) + z^n R(\frac{1}{z}) \\ A^R(z) &= D^R(z)B^R(z) + z^{n-m+1}R^R(z) \end{aligned}$$

由于 $deg(D^R) \le n - m$, 所以可以在模 z^{n-m+1} 的情况下求解 $D^R(z)$, 翻转后带入原方程求出 $R(z)$。

一定要想清楚每个多项式的次数, 以及在模 x 的几次方下计算。

7.5.5 多项式求导与积分

根据 $(x^n)' = nx^{n-1}$ 可得

$$\begin{aligned} F'(z) &= \sum_{i=1}^{n-1} i c_i z^{i-1} \\ \int F(z)\mathrm{d}z &= \sum_{i=1}^{n-1} \frac{c_{i-1}}{i} \cdot z^i \end{aligned}$$

时间复杂度 $O(n)$。

7.5.6 多项式 ln

考虑对 $\ln A(z)$ 求导:

$$(\ln A(z))' = \frac{A'(z)}{A(z)}$$

所以有

$$\ln A(z) = \int \frac{A'(z)}{A(z)}\mathrm{d}z$$

由于要求逆元, 时间复杂度 $O(n \lg n)$。

7.5.7 多项式 exp

先进行如下变换:

$$F(z) - e^{A(z)} \equiv 0 \pmod{z^n} \Rightarrow \ln F(z) - A(z) \equiv 0 \pmod{z^n}$$

直接牛顿迭代法得

$$\begin{aligned} F(z) &= F_0(z) - (\ln F_0(z) - A(z))F_0(z) \\ &= F_0(z)(1 - \ln F_0(z) + A(z)) \end{aligned}$$

一般其输入的常数项为 0, 由麦克劳林级数展开式得常数项恒为 1。

7.5.8 多项式快速幂

使用常规快速幂可以得到 $O(n \lg n \lg k)$ 的复杂度。但是通过如下变形:

$$F^k(z) = e^{k \ln F(z)}$$

使用多项式 ln/exp 可以得到 $O(n \lg n)$ 的复杂度。

注意此种情况只适用于常数项为 1 的情况,其它情况需要缩放多项式。

7.5.9 多项式三角函数

由欧拉公式可得

$$e^{F(z)i} = \cos F(z) + \sin F(z)i$$

在复数域上做多项式 exp。

上述内容基本上使用牛顿迭代法递归求解, 下面的模板(LOJ150 挑战多项式)涵盖了大部分操作:

```
#include <cstdio>
#include <cstring>
#include <vector>
//#define CHECK
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
typedef long long Int64;
#define asInt64 static_cast<Int64>
const int mod = 998244353;
int add(int a, int b) {
    a += b;
    return a < mod ? a : a - mod;
}
int sub(int a, int b) {
    a -= b;
    return a >= 0 ? a : a + mod;
}
Int64 powm(Int64 a, int k) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
namespace Residue {
    Int64 i2;
    struct Complex {
        Int64 a, b;
        Complex(Int64 a, Int64 b) : a(a), b(b) {}
        Complex operator*(const Complex& rhs) const {
            return Complex(
                (a * rhs.a + b * rhs.b % mod * i2) %
                    mod,
                (a * rhs.b + b * rhs.a) % mod);
        }
    };
    Complex powm(Complex a, int k) {
        Complex res(1, 0);
        while(k) {
```

```
            if(k & 1)
                res = res * a;
            k >>= 1, a = a * a;
        }
        return res;
    }
    Int64 getRandom() {
        static int seed = 2523;
        return (seed = seed * 48271LL % 2147483647) %
            mod;
    }
    int sqrt(int x) {
        if(x == 0)
            return 0;
        while(true) {
            Int64 b = getRandom();
            i2 = sub(b * b % mod, x);
            if(::powm(i2, (mod - 1) / 2) == mod - 1) {
                int a = powm(Complex(b, 1),
                             (mod + 1) / 2)
                            .a,
                    b = mod - a;
                return a < b ? a : b;
            }
        }
        return -1;
    }
}
const int size = 1 << 18;
int tot, root[size], invR[size];
void init(int n) {
    tot = n;
    Int64 base = powm(3, (mod - 1) / n);
    Int64 invBase = powm(base, mod - 2);
    root[0] = invR[0] = 1;
    for(int i = 1; i < n; ++i)
        root[i] = root[i - 1] * base % mod;
    for(int i = 1; i < n; ++i)
        invR[i] = invR[i - 1] * invBase % mod;
}
void NTT(int n, int* A, const int* w) {
    for(int i = 0, j = 0; i < n; ++i) {
        if(i < j)
            std::swap(A[i], A[j]);
        for(int l = n >> 1; (j ^= l) < l; l >>= 1)
            ;
    }
    for(int i = 2; i <= n; i <<= 1) {
        int m = i >> 1, fac = tot / i;
        for(int j = 0; j < n; j += i)
```

```
            for(int k = 0; k < m; ++k) {
                int &x = A[j + k + m], &y = A[j + k];
                int t = x * asInt64(w[k * fac]) % mod;
                x = sub(y, t);
                y = add(y, t);
            }
        }
    }
    typedef std::vector<int> Poly;
    void DFT(int n, Poly& A) {
        NTT(n, A.data(), root);
    }
    void IDFT(int n, Poly& A, int rn) {
        NTT(n, A.data(), invR);
        Int64 div = powm(n, mod - 2);
        for(int i = 0; i < rn; ++i)
            A[i] = A[i] * div % mod;
        memset(A.data() + rn, 0, sizeof(int) * (n - rn));
    }
    void copy(Poly& dst, const Poly& src, int siz) {
        memcpy(dst.data(), src.data(), sizeof(int) * siz);
    }
    void reset(Poly& x) {
        memset(x.data(), 0, sizeof(int) * x.size());
    }
    int getSize(int x) {
        x <<= 1;
        int p = 1;
        while(p < x)
            p <<= 1;
        return p;
    }
    void inv(int n, const Poly& sf, Poly& g) {
        if(n == 1)
            g[0] = powm(sf[0], mod - 2);
        else {
            int h = (n + 1) >> 1;
            inv(h, sf, g);
            int p = getSize(n);
            DFT(p, g);

            Poly f(p);
            copy(f, sf, n);
            DFT(p, f);

            for(int i = 0; i < p; ++i) {
                g[i] = (2 - asInt64(g[i]) * f[i]) % mod *
                    g[i] % mod;
                if(g[i] < 0)
                    g[i] += mod;
```

```
        }
        IDFT(p, g, n);
#ifdef CHECK
        for(int i = 0; i < n; ++i) {
            int sum = 0;
            for(int j = 0; j <= i; ++j)
                sum =
                    (sum + asInt64(sf[j]) * g[i - j]) %
                    mod;
            if(i == 0 && sum != 1)
                throw;
            if(i != 0 && sum != 0)
                throw;
        }
#endif
    }
}
Int64 inv2;
void sqrt(int n, const Poly& sf, Poly& g) {
    if(n == 1)
        g[0] = Residue::sqrt(sf[0]);
    else {
        int h = (n + 1) >> 1;
        sqrt(h, sf, g);

        int p = getSize(n);
        Poly f(p);
        copy(f, sf, n);
        DFT(p, f);

        Poly tg(p), ig(p);
        copy(tg, g, h);
        inv(n, tg, ig);
        tg.clear();
        DFT(p, ig);
        DFT(p, g);

        for(int i = 0; i < p; ++i)
            g[i] = inv2 *
                (g[i] + asInt64(f[i]) * ig[i] % mod) %
                mod;
        IDFT(p, g, n);
#ifdef CHECK
        for(int i = 0; i < n; ++i) {
            int sum = 0;
            for(int j = 0; j <= i; ++j)
                sum =
                    (sum + asInt64(g[j]) * g[i - j]) %
                    mod;
            if(sum != sf[i])
```

```
                throw;
            }
    #endif
        }
    }
    void der(int n, Poly& f) {
        for(int i = 1; i < n; ++i)
            f[i - 1] = i * asInt64(f[i]) % mod;
        f[n - 1] = 0;
    }
    int lut[100005];
    void preInv(int n) {
        static int cur = 1;
        while(cur < n) {
            ++cur;
            lut[cur] = asInt64(mod - mod / cur) *
                lut[mod % cur] % mod;
        }
    }
    void inte(int n, Poly& f) {
        preInv(n);
        for(int i = n; i >= 1; --i)
            f[i] = asInt64(lut[i]) * f[i - 1] % mod;
        f[0] = 0;
    }
    void ln(int n, Poly& sf, Poly& g) {
        int p = getSize(n);
        inv(n, sf, g);
        DFT(p, g);
        der(n, sf);
        DFT(p, sf);
        for(int i = 0; i < p; ++i)
            g[i] = asInt64(sf[i]) * g[i] % mod;
        IDFT(p, g, n);
        inte(n, g);
    }
    void exp(int n, const Poly& sf, Poly& g) {
        if(n == 1)
            g[0] = 1;
        else {
            int h = (n + 1) >> 1;
            exp(h, sf, g);

            int p = getSize(n);
            Poly tg(p), lng(p);
            copy(tg, g, h);
            ln(n, tg, lng);
            tg.clear();
            DFT(p, lng);
            DFT(p, g);
```

```
250
            Poly f(p);
            copy(f, sf, n);
            DFT(p, f);

            for(int i = 0; i < p; ++i) {
                g[i] = g[i] * asInt64(1 - lng[i] + f[i]) %
                    mod;
                if(g[i] < 0)
                    g[i] += mod;
260         }
            IDFT(p, g, n);
        }
    }
    int main() {
        lut[1] = 1;
        inv2 = powm(2, mod - 2);
        int n = read() + 1;
        Int64 k = read();
        int end = n + 1;
270     int p = getSize(end);
        init(p);
        Poly in(end), a(p), b(p);
        for(int i = 0; i < n; ++i)
            in[i] = read();
        sqrt(end, in, a);
        inv(end, a, b);
        inte(end, b);
        reset(a);
        exp(end, b, a);
280     in[0] = 2;
        for(int i = 0; i < n; ++i)
            a[i] = sub(in[i], a[i]);
        reset(b);
        ln(end, a, b);
        b[0] = 1;
        reset(a);
        ln(end, b, a);
        for(int i = 0; i < end; ++i)
            a[i] = k * a[i] % mod;
290     reset(b);
        exp(end, a, b);
        der(end, b);
        end = n - 1;
        for(int i = 0; i < end; ++i)
            printf("%d ", b[i]);
        return 0;
    }
```

7.5.10 进制转换

将 A 进制数转换为 B 进制数。

A 进制数可表示为 $\sum_{i=0}^{n} a_i A^i$, 求出其在 B 进制下的值。

对其分治, 即

$$\sum_{i=0}^{n} a_i A^i = \sum_{i=0}^{\frac{n}{2}-1} a_i A^i + A^{\frac{n}{2}} \sum_{i=0}^{\frac{n}{2}} a_{i+\frac{n}{2}} A^i$$

预处理出 A^i 对应的 B 进制数, 然后分治, 同时使用 FFT 优化右边的卷积。

时间复杂度 $O(n \lg^2 n)$。

7.5.11 多项式多点求值

已知 $A(z)$ 与 n 个点 $z_0, z_1, \cdots, z_{n-1}$, 求 $A(z_0), A(z_1), \cdots, A(z_{n-1})$。

将要求的 x 分为两半, 那么左右两半对应的插值多项式的次数为 $[\frac{n}{2}]$, 求出这两个插值多项式后递归计算。

对于左半部分, 考虑多项式 $B(z) = \prod_{i=0}^{[\frac{n}{2}]}(z - z_i)$, 满足 $deg(B) = [\frac{n}{2}]$。令 $A(z) = D(z)B(z) + R(z)$, 当 z 为左半部分中的点时, $D(z)B(z)$ 为 0, 即 $A(z) = R(z)$。那么可以两边同时模 $B(z)$, 达到次数减半的效果。右边部分类似。

注意 $B(z)$ 要 $O(n \lg^2 n)$ 分治预处理, 空间 $O(n \lg n)$, 递归树根的多项式不必计算。

时间复杂度 $O(n \lg^2 n)$。

代码:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int mod = 998244353;
typedef long long Int64;
#define asInt64 static_cast<Int64>
Int64 powm(Int64 a, int k) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
```

```
    }
    return res;
}
int add(int a, int b) {
    a += b;
    return a < mod ? a : a - mod;
}
int sub(int a, int b) {
    a -= b;
    return a >= 0 ? a : a + mod;
}
typedef std::vector<int> Poly;
Poly rc[20], irc[20], rvc[20];
const Poly& getRev(int k) {
    if(rvc[k].empty()) {
        int n = 1 << k, off = k - 1;
        Poly& rev = rvc[k];
        rev.resize(n);
        for(int i = 1; i < n; ++i)
            rev[i] =
                rev[i >> 1] >> 1 | ((i & 1) << off);
    }
    return rvc[k];
}
const Poly& getRoot(int k) {
    if(rc[k].empty()) {
        int n = 1 << k;
        Int64 base = powm(3, (mod - 1) / n);
        Poly& root = rc[k];
        root.resize(n);
        root[0] = 1;
        for(int i = 1; i < n; ++i)
            root[i] = root[i - 1] * base % mod;
    }
    return rc[k];
}
const Poly& getInvRoot(int k) {
    if(irc[k].empty()) {
        int n = 1 << k;
        Int64 base = powm(3, (mod - 1) / n);
        Int64 invBase = powm(base, mod - 2);
        Poly& invR = irc[k];
        invR.resize(n);
        invR[0] = 1;
        for(int i = 1; i < n; ++i)
            invR[i] = invR[i - 1] * invBase % mod;
    }
    return irc[k];
}
typedef const Poly& (*Func)(int);
```

```
void NTT(int n, Poly& A, Func w) {
    int p = 0;
    while((1 << p) != n)
        ++p;
    const Poly& rev = getRev(p);
    for(int i = 0; i < n; ++i)
        if(i < rev[i])
            std::swap(A[i], A[rev[i]]);
    for(int i = 2, cp = 1; i <= n; i <<= 1, ++cp) {
        int m = i >> 1;
        const Poly& cw = w(cp);
        for(int j = 0; j < n; j += i)
            for(int k = 0; k < m; ++k) {
                int &x = A[j + k], &y = A[j + m + k];
                int t = asInt64(cw[k]) * y % mod;
                y = sub(x, t);
                x = add(x, t);
            }
    }
}
int* data(Poly& A) {
    return &*A.begin();
}
const int* data(const Poly& A) {
    return &*A.begin();
}
void DFT(int n, Poly& A) {
    NTT(n, A, getRoot);
}
void IDFT(int n, Poly& A, int b, int e,
          bool clear = true) {
    NTT(n, A, getInvRoot);
    Int64 div = powm(n, mod - 2);
    for(int i = b; i < e; ++i)
        A[i] = A[i] * div % mod;
    if(clear) {
        memset(data(A), 0, sizeof(int) * b);
        memset(data(A) + e, 0, sizeof(int) * (n - e));
    }
}
void copyPoly(Poly& dst, const Poly& src, int siz) {
    memcpy(data(dst), data(src), sizeof(int) * siz);
}
void invImpl(int n, const Poly& sf, Poly& g) {
    if(n == 1)
        g[0] = powm(sf[0], mod - 2);
    else {
        int h = n >> 1;
        invImpl(h, sf, g);
```

```
        Poly dftg(n);
        copyPoly(dftg, g, h);
        DFT(n, dftg);

        Poly f(n);
        copyPoly(f, sf, n);
        DFT(n, f);
        for(int i = 0; i < n; ++i)
            f[i] = asInt64(f[i]) * dftg[i] % mod;
        IDFT(n, f, h, n);

        DFT(n, f);
        for(int i = 0; i < n; ++i)
            f[i] = asInt64(f[i]) * dftg[i] % mod;
        IDFT(n, f, h, n, false);

        for(int i = h; i < n; ++i)
            g[i] = (f[i] ? mod - f[i] : 0);
    }
}
void inv(int n, const Poly& sf, Poly& g) {
    int p = 1;
    while(p < n)
        p <<= 1;
    invImpl(p, sf, g);
    memset(data(g) + n, 0, sizeof(int) * (p - n));
}
Poly doMod(const Poly& A, const Poly& B) {
    if(A.size() < B.size())
        return A;
    if(A.size() < 100) {
        const Int64 fac = mod - 1;
        Poly res(A);
        for(int i = res.size() - 1; i >= B.size() - 1;
            --i)
            if(res[i]) {
                Int64 k = res[i] * fac % mod;
                for(int j = B.size() - 1, cp = i;
                    j >= 0; --j, --cp)
                    res[cp] =
                        (res[cp] + k * B[j]) % mod;
            }
        res.resize(B.size() - 1);
        return res;
    }
    int p = 1;
    while(p < 2 * A.size())
        p <<= 1;
    Poly AR(p);
    int siz = A.size() - B.size() + 1;
```

```
        std::reverse_copy(A.end() - siz, A.end(),
                          AR.begin());
        Poly BR(p);
        std::reverse_copy(
            B.end() -
                std::min(static_cast<int>(B.size()), siz),
180         B.end(), BR.begin());
        Poly IBR(p);
        inv(siz, BR, IBR);

        DFT(p, AR);
        DFT(p, IBR);
        Poly C(p);
        for(int i = 0; i < p; ++i)
            C[i] = asInt64(AR[i]) * IBR[i] % mod;
        IDFT(p, C, 0, siz);
190     std::reverse(C.begin(), C.begin() + siz);
        p >>= 1;

        Poly dftB(p);
        copyPoly(dftB, B, B.size());
        DFT(p, dftB);
        DFT(p, C);
        for(int i = 0; i < p; ++i)
            C[i] = asInt64(C[i]) * dftB[i] % mod;
        IDFT(p, C, 0, B.size() - 1, false);
200     Poly res(B.size() - 1);
        for(int i = 0; i < res.size(); ++i)
            res[i] = sub(A[i], C[i]);
        return res;
    }
    const int size = 64005;
    Poly A[size << 2];
    void pre(int l, int r, int id) {
        if(l == r) {
            Poly& P = A[id];
            P.resize(2);
210         int x = read();
            P[0] = (x ? mod - x : 0);
            P[1] = 1;
        } else {
            int m = (l + r) >> 1;
            pre(l, m, id << 1);
            pre(m + 1, r, id << 1 | 1);
            if(id == 1)
                return;
220         int p = 1, siz = r - l + 1;
            while(p <= siz)
                p <<= 1;
            Poly& P = A[id];
```

```
        P.resize(siz + 1);
        Poly X(p), Y(p);
        copyPoly(X, A[id << 1], A[id << 1].size());
        copyPoly(Y, A[id << 1 | 1],
                 A[id << 1 | 1].size());
        DFT(p, X);
        DFT(p, Y);
        for(int i = 0; i < p; ++i)
            X[i] = asInt64(X[i]) * Y[i] % mod;
        IDFT(p, X, 0, siz + 1, false);
        copyPoly(P, X, siz + 1);
    }
}
int res[size];
void solve(int l, int r, int id, const Poly& X) {
    if(l == r)
        res[l] = X[0];
    else {
        int m = (l + r) >> 1;
        solve(l, m, id << 1, doMod(X, A[id << 1]));
        solve(m + 1, r, id << 1 | 1,
              doMod(X, A[id << 1 | 1]));
    }
}
int main() {
    int n = read();
    int m = read();
    Poly A(n + 1);
    for(int i = 0; i <= n; ++i)
        A[i] = read();
    pre(1, m, 1);
    solve(1, m, 1, A);
    for(int i = 1; i <= m; ++i)
        printf("%d\n", res[i]);
    return 0;
}
```

7.5.12 多项式多点插值

对待插值点分治, 假设求出了左半部分插值多项式 $A_{left}(z)$, 构造出左半部分的 $B(z)$ 使其满足 $A(z) = A_{left}(z) + A_{right}(z)B(z)$, 那么左半部分的点都将在 $A(z)$ 上。仅考虑右半部分:

$$\forall(x_i, y_i) \in P_{right}, y_i = A_{left}(x_i) + A_{right}(x_i)B(x_i)$$

化简得

$$A_{right}(x_i) = \frac{y_i - A_{left}(x_i)}{B(x_i)}$$

那么可利用多点求值得到一组新的插值点, 递归求解。时间复杂度 $O(n \lg^3 n)$。

7.5.13 组合数取模

求 $\binom{n}{m} \bmod p$ 的值, 其中 p 为素数, $n, m \leq 1e9$。

- $p \leq 1e6$:线性预处理逆元 +lucas;
- $n < p$:可知 $n!$ 与 p 互质, 因此只要求 $n!$。可构造多项式:

$$\begin{aligned} Q(x) &= \prod_{i=1}^{\sqrt{n}} (x+i) \\ n! &= \prod_{i=0}^{\sqrt{n}-1} Q(i\sqrt{n}) \end{aligned}$$

  卷积出 $Q(x)$ 后在 $i\sqrt{n} = 0, \sqrt{n}, \cdots, (\sqrt{n}-1)\sqrt{n}$ 上多点求值, 时间复杂度 $O(\sqrt{n}\lg^2 n)$。(在考场上不太好写, 可以用一些预处理时间每隔一段打表, 查询时定位到对应位置暴力计算, 适用于 n 较大而查询很少的情况。)

- $n \geq p$:结合上述两种做法, 易知第二种做法最多只执行一次。

7.5.14 CDQ 分治 FFT

已知函数 g 在 $[1, n)$ 上的值, 求函数 $f(x) = \sum_{y=1}^{x} f(x-y)g(y)$ 在 $[1, n)$ 上的值, 其中 $f(0) = 1$。

对于这种自我依赖的卷积, 可以考虑将待求值一分为二, 分治求解。**分治到小规模时直接 $O(n^2)$ 暴力求解。**

记卷积过程为 $conv(l, r)$, m 为 l, r 中点, 首先递归调用 $conv(l, m)$ 求出区间的前一半, 然后计算使用卷积累加前一半对后一半的贡献, 然后递归调用 $conv(m+1, r)$ 补足区间后一半的剩余项。时间复杂度 $O(n\lg^2 n)$。

事实上这就是 CDQ 分治的过程, 思路参见第19.1.16节。

CDQ 分治 FFT 时, 需要对整个多项式进行平移, 可能导致不知道该从哪个位置, 哪个方向取卷积结果。这时使用端点来计算对应的位置, 找出对应规律。

7.5.14.1 优化

将 F 与 G 看做序列 f, g 的 OGF, 令 $g(0) = 0$, 计算 $F(x) \otimes G(x) = F(x) - f(0)x^0$, 变形为 $F(x) \equiv \frac{1}{1-G(x)} \pmod{x^n}$, 使用多项式求逆 $O(n\lg n)$ 解决。

7.5.15 多项式乘积(普通分治 FFT)

求多个多项式之积(积的次数上界为 W)可以使用分治乘法解决, 时间复杂度 $O(W\lg W\lg n)$。

7.5.15.1 优化 A

由于求的是多项式乘积, 因此在 IDFT 时不用做除法, 而是记录除法因子, 在最终答案中做 1 次除法。

7.5.15.2 优化 B

使用优先队列维护多项式次数, 然后按照类似赫夫曼编码的策略贪心合并多项式。参见 kczno1 的代码[7]。

7.5.15.3 优化 C

若它为多个一次多项式相乘, 则会出现大量小规模卷积, 使用暴力解决。

以上内容参考了 picks [8] 、Miskcoo[9]和 VictoryCzt[10]的博客。

7.5.16 二元多项式卷积

该需求源自 2015 年国家集训队论文集中金策的《生成函数的运算与组合计数问题》, 用于解决二元生成函数的卷积。

将二元函数的系数排为矩阵, 对每一行做 DFT, 再对每一列做 DFT, 将点值相乘, 对每一列做 IDFT, 每一行做 IDFT。整个过程与 DFT 计算卷积的过程类似。记多项式次数为 n, m, 时间复杂度为 $O(nm \lg nm)$。

7.5.17 循环卷积

该需求来自 CTSC2010 性能优化, 题意为求 ab^C 在模 x^n 意义下的循环卷积模 $n+1$ 的值。

使用常用的基 2NTT, 必须把卷积规模开到 $2n$, 然后每次把溢出部分加回去。即使使用快速幂也带来 $O(\lg n)$ 的复杂度。

不过 NTT 可以使用混合基, 例题有一个性质, 即 n 是 smooth number, 可以分解为 $2^a3^b5^c7^d$ 的形式。那么可以按照当时 FFT 的推导写出基 2, 基 3, 基 5, 基 7 的蝴蝶操作, 然后使用递归形式求解。

由于是循环卷积, 求出点值后可以不考虑溢出, 即无需限制点值乘法的次数, 允许直接在点值上做快速幂。

参考代码:

```
0 #include <cstdio>
  #include <vector>
  namespace IO {
      char in[1 << 24];
      void init() {
          fread(in, 1, sizeof(in), stdin);
      }
      int getc() {
```

[7] https://loj.ac/submission/110667

[8] Newton's Method of Polynomial ≪ Picks's Blog
http://picks.logdown.com/posts/209226-newtons-method-of-polynomial
Positional Notation Conversion ≪ Picks's Blog
http://picks.logdown.com/posts/208342-positional-notation-conversion
Binomial Coefficient Modulo Prime ≪ Picks's Blog
http://picks.logdown.com/posts/245545-binomial-coefficient-modulo-prime

[9] 牛顿迭代法在多项式运算的应用–Miskcoo's Space
http://blog.miskcoo.com/2015/06/polynomial-with-newton-method
多项式除法及求模–Miskcoo's Space
http://blog.miskcoo.com/2015/05/polynomial-division
多项式的多点求值与快速插值–Miskcoo's Space
http://blog.miskcoo.com/2015/05/polynomial-multipoint-eval-and-interpolation

[10] 分治 FFT 学习笔记 https://blog.csdn.net/VictoryCzt/article/details/82939586

```
        static char* S = in;
        return *S++;
    }
    char out[1 << 23], *S = out;
    void putc(char ch) {
        *S++ = ch;
    }
    void uninit() {
        fwrite(out, S - out, 1, stdout);
    }
}
int read() {
    int res = 0, c;
    do
        c = IO::getc();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = IO::getc();
    }
    return res;
}
void write(int x) {
    if(x >= 10)
        write(x / 10);
    IO::putc('0' + x % 10);
}
typedef long long Int64;
typedef std::vector<int> Poly;
const int size = 500005;
int pw[size], n, mod;
const int P[4] = { 2, 3, 5, 7 };
void FFT(Poly& A);
void FFTImpl(Poly& A, int p) {
    Poly C[7];
    int csiz = A.size() / p;
    for(int i = 0; i < p; ++i) {
        C[i].resize(csiz);
        for(int j = i, k = 0; j < A.size();
            j += p, ++k)
            C[i][k] = A[j];
        FFT(C[i]);
    }
    int off = n / A.size();
    for(int i = 0; i < p; ++i) {
        int base = i * csiz;
        for(int j = 0; j < csiz; ++j) {
            Int64 res = 0, fac = 1, dst = base + j,
                  mul = pw[off * dst];
            for(int k = 0; k < p; ++k) {
```

```
                res = (res + fac * C[k][j]) % mod;
                fac = fac * mul % mod;
            }
            A[dst] = res;
        }
    }
}
void FFT(Poly& A) {
    if(A.size() != 1) {
        for(int i = 0; i < 4; ++i)
            if(A.size() % P[i] == 0)
                return FFTImpl(A, P[i]);
    }
}
Int64 powm(Int64 a, int k) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
int main() {
    IO::init();
    n = read();
    mod = n + 1;
    int k = read();

    Int64 g = 0;
    for(int i = 2; i <= n; ++i) {
        bool flag = true;
        for(int j = 0; j < 4; ++j)
            if(n % P[j] == 0 &&
               powm(i, n / P[j]) == 1) {
                flag = false;
                break;
            }
        if(flag) {
            g = i;
            break;
        }
    }
    pw[0] = 1;
    for(int i = 1; i < n; ++i)
        pw[i] = pw[i - 1] * g % mod;

    Poly A(n);
    for(int i = 0; i < n; ++i)
        A[i] = read();
```

```
    FFT(A);

    Poly B(n);
    for(int i = 0; i < n; ++i)
        B[i] = read();
    FFT(B);

    Poly C(n);
    for(int i = 0; i < n; ++i)
        C[i] = A[i] * powm(B[i], k) % mod;
    Int64 ig = powm(g, mod - 2);
    for(int i = 1; i < n; ++i)
        pw[i] = pw[i - 1] * ig % mod;
    FFT(C);

    Int64 div = n;
    for(int i = 0; i < n; ++i) {
        write(C[i] * div % mod);
        IO::putc('\n');
    }
    IO::uninit();
    return 0;
}
```

该内容参考了 skywalkert 的博客[11]。

对于没有特殊性质的 n，若要求模 x^n 意义下的循环卷积，可以使用 Bluestein's Algorithm。参见国家集训队 2016 论文集毛啸的《再探快速傅里叶变换》。

7.5.18 CZT

记 ω 为主 n 次单位根，DFT 求的是：

$$B_i = \sum_{k=0}^{n-1} \omega^{ki} A_k$$

尝试将其变形：

$$\begin{aligned} B_i &= \sum_{k=0}^{n-1} \omega^{ki} A_k \\ &= \sum_{k=0}^{n-1} \omega^{-\frac{(i-k)^2 - i^2 - k^2}{2}} A_k \\ &= \omega^{-i^2/2} \sum_{k=0}^{n-1} \omega^{-(i-k)^2/2} \omega^{-k^2/2} A_k \end{aligned}$$

如此便将 DFT 表示为类似卷积的形式，唯一不足在于求和上下界与卷积不同。

[11] BZOJ 1919 [Ctsc2010] 性能优化
https://blog.csdn.net/skywalkert/article/details/51737272

继续变形, 记 $C_i = \omega^{-(i-n)^2/2}$:

$$\begin{aligned}\omega^{i^2/2}a_i = D_{i+n} &= \sum_{k=0}^{n-1}\omega^{-(i-k)^2/2}\omega^{-k^2/2}A_k \\ &= \sum_{k=0}^{n-1}C_{i+n-k}\omega^{-k^2/2}A_k \\ &= \sum_{k=0}^{i+n}C_{i+n-k}\omega^{-k^2/2}A_k\end{aligned}$$

该变形利用了 A_k 只在 $[0, n-1]$ 处有贡献的性质。

如此便可以使用 FFT 计算循环卷积, 但卷积规模大了一倍。

事实上 CZT 是 DFT 的广义形式, 注意到 ω 的单位根性质并没有被用到, 可以使用其它数代替。

该算法参考了国家集训队 2016 论文集毛啸的《再探快速傅里叶变换》。

7.6 多项式算法封装

牛顿迭代法/分治 FFT 时使用。

使用 vector 存储多项式。

```
0 typedef std::vector<int> Poly;
```

封装一个工具函数计算做乘法所需最小规模。

```
0 int getSize(int x) {
      x <<= 1;
      int p = 1;
      while(p < x)
          p <<= 1;
      return p;
  }
```

封装一个工具函数用于拷贝多项式, 参数顺序遵循 memcpy。

```
0 void copyPoly(Poly& dst, const Poly& src, int siz) {
      memcpy(dst.data(), src.data(), sizeof(int) * siz);
  }
```

原来的函数名称是 copy, 但是编译器经常将其与 std::copy 在重载决议中混淆(比如 siz 的实参类型为 std::size 时)。并且涉及模板和 STL 的代码并不好根据编译器错误信息发现错误位置。

封装 DFT、IDFT, 注意再给 IDFT 一个参数指示模数, 只对需要的系数做除法。

```
0 void DFT(int n, Poly& A) {
      NTT(n, A.data(), root);
  }
  void IDFT(int n, Poly& A, int rn) {
      NTT(n, A.data(), invR);
      Int64 div = powm(n, mod - 2);
      for(int i = 0; i < rn; ++i)
```

```
        A[i] = A[i] * div % mod;
    memset(A.data() + rn, 0, sizeof(int) * (n - rn));
}
```

牛顿迭代法系列函数一般使用如下签名:

```
void func(int n, const Poly& sf, Poly& g)
```

其中 sf 为原多项式, 使用时拷贝, g 尽量重复使用, 空间由调用端分配。
以多项式求逆为例:

```
void inv(int n, const Poly& sf, Poly& g) {
    if(n == 1)
        g[0] = powm(sf[0], mod - 2);
    else {
        int h = (n + 1) >> 1;
        inv(h, sf, g);
        int p = getSize(n);
        DFT(p, g);

        Poly f(p);
        copyPoly(f, sf, n);
        DFT(p, f);

        for(int i = 0; i < p; ++i) {
            g[i] = (2 - asInt64(g[i]) * f[i]) % mod *
                g[i] % mod;
            if(g[i] < 0)
                g[i] += mod;
        }
        IDFT(p, g, n);
#ifdef CHECK
        for(int i = 0; i < n; ++i) {
            int sum = 0;
            for(int j = 0; j <= i; ++j)
                sum =
                    (sum + asInt64(sf[j]) * g[i - j]) %
                    mod;
            if(i == 0 && sum != 1)
                throw;
            if(i != 0 && sum != 0)
                throw;
        }
#endif
    }
}
```

求逆/开方等操作在调试时最好 $O(n^2)$ 验证其正确性。
简单多项式乘法根据规模使用暴力/FFT 解决。
注意 std::vector<>::data() 是 c++11 新增的用法,使用时注意程序可移植性。

7.7 计算形式幂级数的牛顿迭代法的常数优化

该内容基于 negiizhao 的博客[12]。

一般在多项式开根及 exp 等常数较大的计算中使用, 简单的计算仍然使用原算法。~~事实上下面的算法更简单~~。

记长度为 $2n$ 的 DFT/IDFT 的计算时间为 $T(n)$, n 次形式幂级数乘法的时间为 $(3+o(1))T(n)$, 下面算法的运行时间均以 $T(n)$ 为基准衡量。运行时间有理论上证明与实际测试结果支持。测试用模数为 998244353, 测试规模为 2^{22}, 每个算法重复运行 100 次。以下常数大小证明假装 $T(n)=2T(n/2)$, 即不考虑 lg 的增长。~~所以可以发现实验常数比理论常数更低~~。

7.7.1 卷积

牛顿迭代法需要遇到不同规模的卷积, 原先的卷积实现使用动态计算位逆序和利用消去引理定位对应的单位根。既然我们已经知道了卷积的最大规模, 可以使用最大规模 2 倍的空间预处理出每种规模的单位根和位逆序数组。这样在访问单位根时对缓存友好。实验表明此法的速度是原先的 2 倍, 且代码几乎不需要改动(仅修改 init 与 NTT 部分, 封装的魅力!!!)。由于本节主要注重牛顿迭代法本身的优化, 下面的算法无论是否优化, 都使用优化后的 NTT 作为基础设施。不过为了体现性能提升的重要性, 未优化的算法使用的子过程也是未优化的。

7.7.2 求逆

7.7.2.1 原算法

原算法在每次迭代中执行 2 次 DFT,1 次 IDFT,长度均为 $2n$ 级别。记规模为 n 的多项式求逆时间复杂度为 $A(n)$,有 $A(n)=A(n/2)+(3+o(1))T(n)$,解得 $A(n)=(6+o(1))T(n)$。事实上, 从模 x^n 推到 x^{2n} 的过程的复杂度等于求模 x^n 意义的复杂度。

7.7.2.2 优化

注意到我们已经有了 g 的末 h 项, 没有必要再计算一遍。记子过程求出的结果为 g_0, 有 $fg_0\equiv 1 \pmod{x^h}$。考虑最初牛顿迭代法的计算过程, 迭代式为 $g'=g_0-(fg_0-1)g_0$, 尝试按照表达式计算而不是化简计算。那么一共需要 2 次卷积, 一次为 fg_0, 另一次为 $(fg_0-1)g_0$。

注意这两个卷积有特殊的性质, 假若 n 对齐到 2 的幂, 且卷积规模为 n 而不是 $2n$, 卷积时会导致高次项溢出。根据 FFT 的性质, 溢出部分将右移 $2n$ 次加到原来的结果中, 即 FFT 乘法做的是循环卷积。如果是一般多项式的乘法, 肯定无法将这两部分拆开。记函数 A 的低 h 次项为 A_L, 高 h 次项右移 h 次为 A_H。那么子过程返回的答案满足 $f_Lg_0\equiv 1 \pmod{x^h}, g'_L=g_0$。那么有 $(f_L+f_Hx^h)g_0=f_Lg_0+f_Hg_0x^h$, 在规模为 n 的卷积下, 低 h 项由 f_Lg_0 的低 h 项与 f_Hg_0 的高 h 项组成, 高 h 项由 f_Lg_0 的高 h 项与 f_Hg_0 的低 h 项组成, 由于已知 f_Lg_0 的低 h 项为 1, 很容易分离出溢出的部分, 不过溢出部分在模 x^n 意义下是不需要的。分离出溢出部分后, 注意到接下来要用的是 $fg_0-1 \bmod x^n$, 其低 h 项全为 0。综上所述, 我们只需要做:求 f 与 g_0 在模 x^n 意义下的循环卷积, 然后将低 h 项值 0。

接下来需要计算 $(fg_0-1)g_0 \bmod x^n$, 而 fg_0-1 的低 h 项全为 0, 故最后不需要按照式子做减法, 而是求出该式的高 h 项, 取反后拷贝到 g' 的高 h 项。注意到 $(fg_0-1)_H$ 与 g_0 卷积时, 高 h 项全部被放到卷积结果的低 h 项, 而我们需要的部分对应于其低 h 项, 对

[12] ~~noip 退役选手的一些扯淡~~关于优化形式幂级数计算的牛顿法的常数
http://negiizhao.blog.uoj.ac/blog/4671

应于卷积结果的高 h 项。综上所述, 我们只需要做: 求 fg_0-1 与 g_0 在模 x^n 意义下的循环卷积, 然后把高 h 项的值取反拷贝到 g' 的高 h 项, g' 的低 h 项就是 g_0。

注意到两次卷积都有 g_0, 可以重复使用 $DFT(g_0)$。最终这个算法需要 3 次 DFT 与 2 次 IDFT, 且卷积规模都是 n, 时间复杂度 $(5+o(1))T(n)$。

注意输入数组也要对齐到 2 的幂, 否则会导致 RE 或者漫长的 Debug。

7.7.3 平方根

7.7.3.1 原算法

原算法在每次迭代中执行 3 次 DFT, 1 次 IDFT, 1 次求逆。从模 x^n 推到模 x^{2n} 的时间复杂度为 $(20+o(1))T(n)$, 求模 x^n 意义下的答案也是如此。

7.7.3.2 优化

注意到每次迭代都要求逆, 但是每次求逆的规模也是倍增的。考虑在维护 $f^{1/2}$ 时同时维护 $f^{-1/2}$。每次计算出 $f^{1/2}$ 后使用多项式求逆的迭代方法计算 $f^{-1/2}$。

当然循环卷积同样适用于求平方根的迭代。注意到 $g_0^2-f\equiv 0 \pmod{x^h}$, 在模 x^n 意义下只需取 g_0^2 卷积结果的高 h 项与 f 做减法, 其余项置零, 然后与 g_0^{-1} 卷积, 取高 h 项乘以 $-1/2$ 赋值到 g' 的高 h 项。开方与求逆分别 2 次 DFT 与 2 次 IDFT, 同时共享一次对 g_0^{-1} 的 DFT。时间复杂度为 $(9+o(1))T(n)$。若无必要, 最后一次迭代不需要求逆, 时间复杂度为 $(7+o(1))T(n)$。

7.7.4 指数

7.7.4.1 原算法

原算法的 ln 部分有一次求逆, 一次乘法。迭代其余部分有 3 次 DFT, 1 次 IDFT。时间复杂度 $(26+o(1))T(n)$。

7.7.4.2 优化

注意到 ln 部分需要用到求逆, 考虑迭代的同时维护逆, 再加上循环卷积优化。

但是 ln 部分的求逆精度要达到模 x^n, 而子过程给的精度是模 x^h。因此需要对原迭代式进行变形, 降低精度要求。记 $h_0=g_0^{-1} \bmod x^h$, $f_0=f \bmod x^h$。首先将 f 丢入积分中, 然后注意到 g_0'/g_0-f' 的低 h-1 项为 0, 我们需要的是它的高 $h\sim n-1$ 项, 考虑引入 g_0h_0 以消去 g_0^{-1}。因为 g_0h_0 的低 h 项为 1, 所以乘上 g_0h_0 后需要的部分仍被保留, 且 g_0h_0 的高 h 项无影响。再利用 $g_0h_0-1\equiv 0 \pmod{x^h}$ 减小卷积规模。

$$
\begin{aligned}
g' &\equiv g_0-(\int (g_0'/g_0)-f)g_0 \\
&\equiv g_0-g_0\int (g_0'/g_0-f') \\
&\equiv g_0-g_0\int ((g_0h_0)(g_0'/g_0-f')) \\
&\equiv g_0-g_0\int (h_0g_0'-f'(g_0h_0)) \\
&\equiv g_0-g_0\int (h_0g_0'-f'(g_0h_0-1)-f') \\
&\equiv g_0-g_0(\int (h_0g_0'-f_0'(g_0h_0-1))-f) \pmod{x^n}
\end{aligned}
$$

时间复杂度为 $(12+o(1))T(n)$。

7.7.4.3 CDQ 分治 FFT 计算多项式 exp

事实上使用非牛顿迭代法计算 exp 也是挺不错的, 哪怕它的理论复杂度为 $O(n\lg^2 n)$。在 _rqy 的博客里发现了一个简单的多项式 exp 算法[13]。

实测该方法未优化时在小规模数据 (2^{16}) 下性能稍逊于普通 exp, 但是它简单好写。使用小规模暴力卷积与循环卷积优化后性能仅次于优化后的 exp, 而两者的代码量却约为 1:2 (45:91)。

原理: 记 $G(x)=e^{F(x)}$, 注意到 $(e^{F(x)})'=F'(x)e^{F(x)}$, 那么有 $e^{F(x)}=\int\frac{xF'(x)e^{F(x)}}{x}$, 记 $H(x)=xF'(x)$, 很容易预处理。然后 CDQ 分治计算 $H(x)$ 与 $G(x)$ 的卷积, 在递归到底层时做积分。注意要特判常数项为 1。上下同乘一个 x 是为了求导/积分时不用考虑复杂的移位。卷积时由于只取高次项, 低位与溢出位贡献可以被忽略, 对齐后可使用循环卷积优化。

参考代码:

```
Poly conv(const int* A, const int* B, int siz) {
    int p = siz << 1;
    if(siz <= 16) {
        Poly res(p);
        for(int i = 0; i < siz; ++i)
            for(int j = siz - i; i + j < p; ++j)
                res[i + j] = (res[i + j] +
                              asInt64(A[i]) * B[j]) %
                    mod;
        return res;
    }
    Poly X(p), Y(p);
    memcpy(data(X), A, sizeof(int) * siz);
    memcpy(data(Y), B, sizeof(int) * p);
    DFT(p, X);
    DFT(p, Y);
    for(int i = 0; i < p; ++i)
        X[i] = asInt64(X[i]) * Y[i] % mod;
    IDFT(p, X, siz, p, false);
    return X;
}
void getExpCDQImpl(int b, int e, const Poly& sf,
                   Poly& g) {
    if(b + 1 == e) {
        g[b] = (b == 0 ? 1 :
                         asInt64(g[b]) * lut[b] % mod);
    } else {
        int m = (b + e) >> 1, h = m - b;
        getExpCDQImpl(b, m, sf, g);
        Poly X = conv(data(g) + b, data(sf), h);
        for(int i = m; i < e; ++i)
            g[i] = add(g[i], X[i - b]);
        getExpCDQImpl(m, e, sf, g);
```

[13] _rqy's Code Style for OI https://rqy.moe/uncategorized/rqy-s-Code-Style-for-OI/

```
    }
}
void getExpCDQ(int n, const Poly& sf, Poly& g) {
    int p = 1;
    while(p < n)
        p <<= 1;
    Poly cf(p);
    for(int i = 0; i < n; ++i)
        cf[i] = asInt64(sf[i]) * i % mod;
    getExpCDQImpl(0, p, cf, g);
    memset(data(g) + n, 0, sizeof(int) * (p - n));
}
```

7.7.5 优化方法总结与实现细节

这两个方法通用且优化效果显著。

- 对于子过程需要用到求逆的迭代, 可以考虑同步迭代计算逆。
- 牛顿迭代法中分子项模 x^h 恒为 0, 考虑循环卷积减小卷积规模。

鉴于自己经常忘记初始化与初始化规模不够的情况，今后全面使用惰性初始化。
为了适应循环卷积惰性除法与清零的需要, 更新 IDFT 接口:

```
void IDFT(int n, Poly& A, int b, int e,
    bool clear = true) {
    NTT(n, A, getInvRoot);
    Int64 div = mod - (mod - 1) / n;
    for(int i = b; i < e; ++i)
        A[i] = A[i] * div % mod;
    if(clear) {
        memset(data(A), 0, sizeof(int) * b);
        memset(data(A) + e, 0, sizeof(int) * (n - e));
    }
}
```

clear 标志指示是否需要清零低位/高位, 若需要继续循环卷积则清零, 若仅做加减法则不清零。~~原先使用 lc/hc 标志独立控制高低位清零, 后来发现一般只保留一半, 规模较小的空位清零不影响性能~~。

事实上除法操作中 div 不需要使用快速幂求逆元, 注意到卷积规模都是 2 的幂, 而 $\varphi(mod) = mod - 1$ 肯定有 2 的幂的因子, 那么 $(mod - 1)/n$ 可以整除且其意义为 $-n^{-1}$, 取反后可得 $div = mod - (mod - 1)/n$。

7.7.6 实验结果

测试代码参见仓库中的/Review/Polynomial/Perf 文件夹。

算法	测试比值(优化前/优化后, 以 $T(n)$ 为基准)
多项式乘法	-/3.12
多项式求逆	5.89/4.44
多项式平方根	18.87/6.10
多项式指数	24.69/10.66

使用上述优化的 LOJ150 挑战多项式速度提升 5 倍, 从倒数第 4 一跃成为 rank3 (2019.2.20)。

7.8 本章注记

Miskcoo 的博客中有大量非常优秀的 FFT 系列文章 [14]。
HocRiser 对此系列做了全面的总结 [15]。

[14] http://blog.miskcoo.com/tag/fft
[15] http://www.cnblogs.com/HocRiser/p/8207295.html

Chapter 8

线性代数

8.1 高斯消元 253
8.1.1 变换为上三角矩阵 253
8.1.2 求解线性方程组 254
8.1.3 求解逆矩阵 254
8.2 LUP 分解 257
8.2.1 基本原理 257
8.2.2 LUP 分解 257
8.2.3 正向/反向替换 259
8.2.4 LUP 分解求逆矩阵 259
8.3 行列式 262
8.3.1 定义与性质 262
8.3.2 求行列式 263
8.4 线性基 263
8.4.1 插入 263
8.4.2 合并 263
8.4.3 存在性查询 263
8.4.4 最大值 264
8.4.5 最小非 0 值 264
8.4.6 第 k 小值 264
8.5 最小二乘逼近 265
8.6 常系数齐次线性递推 266

8.1 高斯消元

8.1.1 变换为上三角矩阵

高斯消元的思路很简单: 每次通过初等变换消去第 i 行第 i 列下面的项, 最终变换为一个上三角矩阵。

gauss

```
0 typedef double FT;
  const FT eps=1e-8;
```

```
FT A[size][size];
bool gauss(int n) {
    for(int i=1;i<=n;++i) {
        int x=i;
        for(int j=i+1;j<=n;++j)
            if(fabs(A[j][i])>fabs(A[x][i]))
                x=j;
        if(fabs(A[x][i])<eps)return false;
        if(x!=i) {
            for(int j=i;j<=n;++j)
                std::swap(A[i][j],A[x][j]);
        }
        for(int j=i+1;j<=n;++j) {
            FT fac=A[j][i]/A[i][i];
            for(int k=i;k<=n;++k)
                A[j][k]-=fac*A[i][k];
        }
    }
    return true;
}
```

时间复杂度 $O(n^3)$。

8.1.1.1 精度

对于实数运算, 选择绝对值最大数作为主元, 以减小误差。

8.1.1.2 优化

可以观察矩阵的特殊性来优化消元复杂度, 对于方程间联系不大的情况使用迭代解小规模线性方程组。

8.1.2 求解线性方程组

即求解 $Ax = b$ 的向量 x。注意到上三角矩阵的最后一行是一元方程, 解出并带入倒数第二行, 它还是一元方程, 因此可以不断逆推得到所有解。注意对矩阵 A 的操作也要应用到向量 b 上(实质为两边同时乘上初等矩阵)。时间复杂度 $O(n^2)$。

```
for(int i=n;i>=1;--i) {
    FT sum=B[i];
    for(int j=i+1;j<=n;++j)
        sum-=A[i][j]*X[j];
    X[i]=sum/A[i][i];
}
```

8.1.3 求解逆矩阵

设将 A 变换为上三角矩阵的变换矩阵为 P, 求逆矩阵即求解方程 $PAA^{-1} = PI$, 由矩阵乘法的定义可知, 将 A^{-1} 与 PI 按列拆分, 即得到 $(PA)A_i^{-1} = (PI)_i$ 的形式, 按照求解线性方程组的方法解出逆矩阵的每一列。

由于要额外维护 PI, 所以常数较 LUP 分解大。

8.1.3.1 板子

【P4783】【模板】矩阵求逆 - 洛谷 [1]

InvMatGauss

```
#include <algorithm>
#include <cstdio>
#include <limits>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
const int size = 405, mod = 1000000007;
Int64 inv(Int64 a) {
    Int64 res = 1;
    int k = mod - 2;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
int A[size][size], P[size][size], B[size][size];
bool gauss(int n) {
    for(int i = 1; i <= n; ++i) {
        int x = 0;
        for(int j = i; j <= n; ++j)
            if(A[j][i]) {
                x = i;
                break;
            }
        if(x == 0)
            return false;
        if(x != i) {
            for(int j = i; j <= n; ++j)
                std::swap(A[i][j], A[x][j]);
            for(int j = 1; j <= n; ++j)
                std::swap(P[i][j], P[x][j]);
        }
```

[1] https://www.luogu.org/problemnew/show/P4783

```
        Int64 invi = inv(A[i][i]);
        for(int j = i + 1; j <= n; ++j) {
            Int64 fac = A[j][i] * invi % mod;
            for(int k = i; k <= n; ++k)
                A[j][k] =
                    (A[j][k] - A[i][k] * fac) % mod;
            for(int k = 1; k <= n; ++k)
                P[j][k] =
                    (P[j][k] - P[i][k] * fac) % mod;
        }
    }
    return true;
}
const Int64 limit = std::numeric_limits<Int64>::max() -
    asInt64(mod) * mod;
bool needMod(Int64 x) {
    x = (x >= 0 ? x : -x);
    return x >= limit;
}
int main() {
    int n = read();
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            A[i][j] = read();
    for(int i = 1; i <= n; ++i)
        P[i][i] = 1;
    if(gauss(n)) {
        for(int i = 1; i <= n; ++i) {
            for(int j = n; j >= 1; --j) {
                Int64 sum = P[j][i];
                for(int k = j + 1; k <= n; ++k) {
                    sum -= asInt64(A[j][k]) * B[k][i];
                    if(needMod(sum))
                        sum %= mod;
                }
                B[j][i] =
                    sum % mod * inv(A[j][j]) % mod;
            }
        }
        for(int i = 1; i <= n; ++i) {
            for(int j = 1; j <= n; ++j) {
                int res = B[i][j];
                printf("%d ",
                       res >= 0 ? res : res + mod);
            }
            putchar('\n');
        }
    } else
        puts("No Solution");
    return 0;
```

}

时间复杂度 $O(n^3)$。

《线性代数及其应用》中提到使用高斯消元法求逆矩阵的与上文等价的方法：将待求逆矩阵 A 与单位矩阵 I 并为矩阵 $[A\ I]$，然后使用高斯消元法将 A 消为上三角矩阵，再进一步消为单位矩阵，右边的单位矩阵做同样的初等行变换，最后的结果为 $[I\ A^{-1}]$。

证明：设变换矩阵为 P，有 $AP = I$，根据 IMT 可知 $A^{-1} = P$，并且此时矩阵的右半部分为 $IP = P$。

8.2 LUP 分解

LUP 分解的数值稳定性较高斯消元法强。

8.2.1 基本原理

要求解线性方程组 $Ax = b$，对系数矩阵 A 进行 LUP 分解：

$$PA = LU$$

其中 P 为置换矩阵，L 为下三角矩阵，U 为上三角矩阵。

将 $Ax = b$ 左乘 P，得 $PAx = Pb$，然后用 $PA = LU$ 代换得 $LUx = Pb$。设 $y = Ux$，有 $Ly = Pb$，可以使用类似于 8.1.2节 $O(n^2)$ 求解 y，然后再次使用该方法求出 x。

8.2.2 LUP 分解

8.2.2.1 LU 分解

考虑不会出现不需要换主元的情况 (比如对称正定矩阵)，即 $P = I$。

运用矩阵代数将 A 分解：

$$\begin{aligned}
A &= \left[\begin{array}{c|ccc} a_{11} & a_{12} & \cdots & a_{1n} \\ \hline a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{array}\right] \\
&= \begin{bmatrix} a_{11} & w^T \\ v & A' \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & A' - vw^T/a_{11} \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ v/a_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & L'U' \end{bmatrix} \text{（递归分解子矩阵）} \\
&= \begin{bmatrix} 1 & 0 \\ v/a_{11} & L' \end{bmatrix} \begin{bmatrix} a_{11} & w^T \\ 0 & U' \end{bmatrix} \text{(左右矩阵分别为 L, U)} \\
&= LU
\end{aligned}$$

求出矩阵的外围部分与子矩阵后将子矩阵递归分解。

8.2.2.2 LUP 分解实现

设换主元时把第 1 行(因为要递归分解)与第 k 行交换的置换矩阵为 Q, 则 QA 可以进行 LU 分解, 即

$$QA = \begin{bmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{bmatrix}$$

设子矩阵满足 $P'(A' - vw^T/a_{k1}) = L'U'$, 与 Q 相乘得到置换矩阵 P, 即

$$P = \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} Q$$

继续变换:

$$\begin{aligned} PA &= \begin{bmatrix} 1 & 0 \\ 0 & P' \end{bmatrix} \begin{bmatrix} 1 & 0 \\ v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & P' \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & A' - vw^T/a_{k1} \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & P'(A' - vw^T/a_{k1}) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & I_{n-1} \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & L'U' \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ P'v/a_{k1} & L' \end{bmatrix} \begin{bmatrix} a_{k1} & w^T \\ 0 & U' \end{bmatrix} \\ &= LU \end{aligned}$$

理解性记忆:按照《线性代数及其应用》的描述, LU 分解的目的就是使用一系列行倍加变换把 A 化为阶梯形 U, 同时构造单位下三角矩阵 L 使得对 L 施加相同的行变换后变为 I。记行变换矩阵为 P, 则 $PL = I$ 且 $PA = U$, 由 L 可逆可得 $LP = I$, 继而得到 $A = LU$。行倍加变换的策略就是把每次把当前主元位置下方的元素消为 0, 由于 L 对应的主元位置为 1, 该主元位置下方对应的就是行倍加的系数。

实际操作非常简单:选取非 0 主元置换到对角线上, 然后令 $l_{ji} = a_{ji}/a_{ii}$, 同时将其作为行倍加系数进行行倍加变换。**注意置换时要整行置换。**

```
typedef double FT;
const FT eps=1e-8;
FT A[size][size],B[size];
bool LUP(int n){
    for(int i=1;i<=n;++i) {
        int x=i;
        for(int j=i+1;j<=n;++j)
            if(fabs(A[j][i])>fabs(A[x][i]))
                x=j;
        if(fabs(A[x][i])<eps)return false;
        if(i!=x){
            for(int j=1;j<=n;++j)
                std::swap(A[i][j],A[x][j]);
            std::swap(B[i],B[x]);
        }
```

```
        for(int j=i+1;j<=n;++j) {
            A[j][i]/=A[i][i];
            FT fac=A[j][i];
            for(int k=i+1;k<=n;++k)
                A[j][k]-=fac*A[i][k];
        }
    }
    return true;
}
```

注意 L 与 U 同时覆盖于 A 数组上, 即

$$a_{ij} = \begin{cases} l_{ij} & \text{if } i > j \\ u_{ij} & \text{if } i \le j \end{cases}$$

8.2.3 正向/反向替换

原理同 8.1.2节, 代码如下:

```
FT Y[size],X[size];
void solve(int n) {
    for(int i=1;i<=n;++i) {
        FT sum=B[i];
        for(int j=1;j<i;++j)
            sum-=A[i][j]*Y[j];
        Y[j]=sum;
    }
    for(int i=n;i>=1;--i) {
        FT sum=Y[i];
        for(int j=i+1;j<=n;++j)
            sum-=A[i][j]*X[j];
        X[i]=sum/A[i][i];
    }
}
```

8.2.4 LUP 分解求逆矩阵

同 8.1.3节所述, 求解 n 次线性方程组得到逆矩阵。LUP 分解的好处在于只需维护置换矩阵 P 而不是维护变换矩阵 PI(这两个矩阵的意义不同)。

InvMatLUP

```
#include <algorithm>
#include <cstdio>
#include <limits>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
```

```
        c = getchar();
    }
    return res;
}
const int size = 405, mod = 1000000007;
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
Int64 inv(Int64 a) {
    Int64 res = 1;
    int k = mod - 2;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
int A[size][size], P[size];
bool LUP(int n) {
    for(int i = 1; i <= n; ++i)
        P[i] = i;
    for(int i = 1; i <= n; ++i) {
        int x = 0;
        for(int j = i; j <= n; ++j)
            if(A[i][j]) {
                x = j;
                break;
            }
        if(x == 0)
            return false;
        if(i != x) {
            std::swap(P[i], P[x]);
            for(int j = 1; j <= n; ++j)
                std::swap(A[i][j], A[x][j]);
        }
        Int64 fac = inv(A[i][i]);
        for(int j = i + 1; j <= n; ++j) {
            A[j][i] = A[j][i] * fac % mod;
            Int64 mul = A[j][i];
            for(int k = i + 1; k <= n; ++k)
                A[j][k] =
                    (A[j][k] - mul * A[i][k]) % mod;
        }
    }
    return true;
}
const Int64 limit = std::numeric_limits<Int64>::max() -
    asInt64(mod) * mod;
bool needMod(Int64 x) {
    x = (x >= 0 ? x : -x);
```

```
    return x >= limit;
}
int B[size][size], C[size], D[size];
void solve(int n, int m) {
    for(int i = 1; i <= n; ++i) {
        Int64 sum = (P[i] == m ? 1 : 0);
        for(int j = 1; j < i; ++j) {
            sum -= asInt64(A[i][j]) * C[j];
            if(needMod(sum))
                sum %= mod;
        }
        C[i] = sum % mod;
    }
    for(int i = n; i >= 1; --i) {
        Int64 sum = C[i];
        for(int j = i + 1; j <= n; ++j) {
            sum -= asInt64(A[i][j]) * D[j];
            if(needMod(sum))
                sum %= mod;
        }
        sum %= mod;
        D[i] = sum * inv(A[i][i]) % mod;
    }
    for(int i = 1; i <= n; ++i)
        B[i][m] = D[i];
}
int main() {
    int n = read();
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n; ++j)
            A[i][j] = read();
    if(LUP(n)) {
        for(int i = 1; i <= n; ++i)
            solve(n, i);
        for(int i = 1; i <= n; ++i) {
            for(int j = 1; j <= n; ++j) {
                int val = B[i][j];
                printf("%d ",
                       val >= 0 ? val : val + mod);
            }
            putchar('\n');
        }
    } else
        puts("No Solution");
    return 0;
}
```

以上内容参考了算法导论 [4] 第 28 章矩阵运算。

8.3 行列式

8.3.1 定义与性质

记 $A_{[ij]}$ 为矩阵 A 去掉第 i 行第 j 列后的矩阵, $\tau(P)$ 为序列 P 的逆序对数, $P=(p_1,p_2,\cdots,p_n)$ 为 $1\dots n$ 的排列。那么 $n*n$ 矩阵 A 的行列式定义为:

$$det(A) = \sum_{P=(p_1,p_2,\cdots,p_n)} (-1)^{\tau(P)} \prod_{i=1}^{n} a_{ip_i} \tag{8.1}$$

$$= \begin{cases} a_{11} & \text{if } n=1 \\ \sum_{j=1}^{n} (-1)^{1+j} a_{1j} det(A_{[1j]}) & \text{if } n>1 \end{cases} \tag{8.2}$$

式 8.1可理解为选择 n 个不同行且不同列的元素, 逆序对数为偶数的加, 奇数的减。
式 8.2是式 8.1的递归形式。
记 $A_{ij}=(-1)^{i+j}det(A_{[ij]})$ 为元素 a_{ij} 的**代数余子式**。
行列式拥有如下性质:

性质 8.1 $det(A)=det(A^T)$

证明:根据式 8.1可知, 行和列是无关的, 转置后 $det(A)$ 不变。

性质 8.2 若矩阵 A 的某一行/列为 0, 则 $det(A)=0$。

证明:根据式 8.1可知每个排列中矩阵元素必有一个 0, 所以 $det(A)=0$。

性质 8.3 矩阵 A 的任意一行/列乘以 λ, $det(A)$ 乘以 λ。

证明:根据式 8.1可知每个排列中有一个矩阵元素乘以 λ, 则 $det(A)$ 放大 λ。

性质 8.4 若矩阵 C 的第 i 行/列可分解为 $c=a+b$ 的形式, 则 $det(C)$ 可分解为分别包含这两个向量, 其余元素与矩阵 C 相同的矩阵的行列式之和。

推论 8.5 将矩阵 A 中某一行/列元素加 λ 倍到另一行/列上, $det(A)$ 不变。

将 $det(A')$ 拆分, 有 $det(A')=det(A)+det(A_{add})=det(A)$(因为 A_{add} 有线性相关向量)。

性质 8.6 交换 A 任意两行/列, $det(A)$ 变号。

证明:交换会使逆序对数奇偶性改变(通过计算相邻交换步数可证明), $det(A)$ 变号。

性质 8.7 $det(AB)=det(A)det(B)$

定理 8.8 $det(A)=0 \Leftrightarrow$ 矩阵 A 奇异

证明:$det(A)=0$ 意味着至少有 2 个系数向量是线性相关的, 矩阵 A 是欠定方程组的系数矩阵, 没有逆矩阵。
代数余子式有如下性质:

性质 8.9

$$det(A) = \sum_{j=1}^{n} a_{ij}A_{ij} \ i=1,2,\cdots,n$$

$$= \sum_{i=1}^{n} a_{ij}A_{ij} \ j=1,2,\cdots,n$$

性质 8.10

$$\forall i! = j, \sum_{k=1}^{n} a_{ik} A_{jk} = 0$$

8.3.2 求行列式

定理 8.11 若矩阵 A 为上三角矩阵，则 $det(A)$ 为主对角线上元素之积。

证明：唯一矩阵元素积可能非 0 的排列方案只有在主对角线上。

高斯消元后根据定理 8.11计算行列式，注意在初等变换操作中交换行时要变号**（尤其是在计算模意义下有向生成树计数时）**。

以上内容参考了算法导论 [4] 附录 D 矩阵。

若矩阵比较特殊，有较多的非零项，可以选取 0 的个数最多的一行/列展开。

8.4 线性基

在此仅描述异或线性基。

记数组 b 存储了基中的向量，这是个 01 上三角矩阵。

性质 8.12 线性基内的向量是线性无关的。

通常利用这个性质 + 贪心来解题。

8.4.1 插入

```
int b[bitSize+1];
bool insert(int x) {
    for(int i=bitSize;i>=0;--i)
        if(x&(1<<i)){
            if(b[i])x^=b[i];
            else {
                b[i]=x;
                return true;
            }
        }
    return false;
}
```

按位遍历进行高斯消元，如果被消为 0 则可被原基构造。

8.4.2 合并

暴力将一个线性基内的所有数插到另一个线性基进去。

8.4.3 存在性查询

按位扫描进行高斯消元，如果被消为 0 则在基的张成中。

8.4.4 最大值

```
0 int maxv() {
      int res=0;
      for(int i=bitSize;i>=0;++i)
          res=std::max(res,res^b[i]);
      return res;
  }
```

由于线性基可以张成出整个空间, 因此从高位到低位贪心就能得到最大值。

此外还有一种等价写法, 不使用比较操作, 可以使用于 bitset 上。

```
0 typedef std::bitset<bitSize> Bit;
  Bit maxv() {
      Bit res;
      for(int i=bitSize-1;i>=0;++i)
          if(!res[i])
              res^=b[i];
      return res;
  }
```

8.4.5 最小非 0 值

```
0 int minv() {
      for(int i=0;i<=bitSize;++i)
          if(b[i])
              return b[i];
      return -1;
  }
```

最小值就是最小的基向量。

8.4.6 第 k 小值

首先变换线性基, 使位与位之间独立(即仅 $b[i]$ 含有 $1 << i$), 然后挑出非 0 向量。

```
0 int vb[bitSize+1],vcnt=0;
  void cook() {
      for(int i=bitSize;i>=0;--i)
          for(int j=i-1;j>=0;--j)
              if(b[i]&(1<<j))b[i]^=b[j];
      for(int i=0;i<=bitSize;++i)
          if(b[i])
              vb[vcnt++]=b[i];
  }
```

计算第 k 小时扫描一遍 k, 按 k 的比特位异或对应的基向量。

```
0 int kth(int k) {
      if(k>(1<<vcnt))return -1;
```

```
    int res=0;
    for(int i=0;i<vcnt;++i)
        if(k&(1<<i))
            res^=vb[i];
    return res;
}
```

注意事项

- 要求第 k 大异或和, 输入的 k 要-1。
- 如果要求异或的子集非空, 则需考虑 0 是否能被构造。当且仅当输入各向量**线性无关**时(即输入向量数 =B 中非 0 向量数), 0 不可被构造(仅有平凡解)。因此若 0 可被构造则 k 要-1, 否则 k 不变。

8.5 最小二乘逼近

该方法用来确定基函数的系数以拟合曲线。

设有 m 个数据点 $(x_1,y_1),(x_2,y_2),\cdots,(x_m,y_m)$, n 个基函数 $f_1,f_2,\cdots,f_n$ 记向量 $y=(y_1,y_2.\cdots,y_m)$, 基函数值矩阵 A 为

$$A=\begin{bmatrix} f_1(x_1) & f_2(x_1) & \cdots & f_n(x_1) \\ f_1(x_2) & f_2(x_2) & \cdots & f_n(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_m) & f_2(x_m) & \cdots & f_n(x_m) \end{bmatrix}$$

设系数向量为 c, 则有误差向量 $\eta=Ac-y$。

使用最小二乘的思想, 令 $\|\eta\|^2$ 最小, 即最小化

$$\|Ac-y\|=\sum_{i=1}^{m}\left(\sum_{j=1}^{n}a_{ij}c_j-y_i\right)^2$$

对向量 c 中的每个元素微分并让结果为 0, 即对于单个元素 c_k, 有

$$\frac{\mathrm{d}\|\eta\|^2}{\mathrm{d}c_k}=\sum_{i=1}^{m}2\left(\sum_{j=1}^{n}a_{ij}c_j-y_i\right)a_{ik}=0$$

将 n 个等式结合在一起,发现这是个向量 * 矩阵的形式,可得新的矩阵方程 $(Ac-y)^TA=0$, 记 $A^+=(A^TA)^{-1}A^T$ 为矩阵 A 的伪逆, 有 $c=A^+y$。

以上内容参考了算法导论 [4] 28.3 节。

《线性代数及其应用》中提到了另一种描述方式: 将矩阵 A 看做由 n 个向量张成出的空间, 那么 x 在该空间上的投影为 A^Tx, 要让 Ax 与 y 之间的 "欧几里得距离" 最小, 必须保证 y 的正交投影为 Ax, 那么有 $A^TAx=A^Ty$, 这个方程称为法方程。

更好的理解参见马同学高等数学[2]。

[2]如何理解最小二乘法? https://www.matongxue.com/madocs/818/

8.6 常系数齐次线性递推

给定系数 $a_0, a_2, \cdots, a_k$, 有一个信号 f_n, 满足 k 阶齐次线性差分方程 $\sum_{i=0}^{k} a_i f_{n-i} = 0$ 对所有 n 成立。现在给定信号 f_n 中的连续 k 项, 求信号的任意一项。

一般 a_0 取 1, 其余系数取反, 那么有 $f_n = \sum_{i=1}^{k} a_i f_{n-i}$, 假设给定了 $f_0, f_1, \cdots, f_{k-1}$ 的值, 现在要求出 f_n 的值。

如果 k 足够小, 那么很容易构造转移矩阵, 记向量 F_i 表示 $(f_{i+k-1}, f_{i+k-2}, \cdots, f_i)$, 很容易构造 $k*k$ 的转移矩阵 A, 其中 i 向 $i+1$ 转移系数为 1, i 向 1 转移系数为 a_i。即 $A[i+1][i] = 1, A[1][i] = a_i$, 那么有 $F_n = A^n F_0$。使用矩阵快速幂可得到 $O(k^3 \lg n)$ 的算法。

注意原等式左右向量取第 k 项仍然成立, 即 $(F_n)_{[k]} = (A^n F_0)_{[k]}$。左边就是 f_n, 右边就是以 A^n 的第 k 行为系数的 $f_{0,\cdots,k-1}$ 的线性组合。记这些系数为 $c_{0,\cdots,k-1}$, 同样将 $f_{0,\cdots,k-1}$ 表示为 $A^n F_0$ 的形式, 有 $A^n = \sum_{i=0}^{k-1} c_i A^i$, 记右式为矩阵 A 的多项式表达 $R(M)$。设存在矩阵多项式 $F(M), G(M)$, 满足 $G(M)$ 的次数为 k, $F(M)G(M)$ 的次数为 n, 且 $M^n = F(M)G(M) + R(M)$。根据 Cayley-Hamilton 定理, 若 $G(M)$ 为矩阵 A 的特征多项式, 其次数恰好为 k 且 $G(A) = 0$。由于 $G(M)$ 的次数为 k, $R(M) \equiv M^n \pmod{G(M)}$, 多项式取模可求出 $R(M)$。并且由于 $G(A) = 0$, 此时有 $A^n = R(A)$。拿到 $R(M)$ 后就可以直接 $O(k)$ 求值。

接下来讨论如何构造出矩阵 A 的特征多项式 $G(M)$。根据定义有 $G(\lambda) = \det(\lambda I_k - A)$, 多项式高斯消元求行列式十分麻烦。注意到矩阵 $\lambda I_k - A$ 的特殊性, 在第一行展开求和, 去除第一行第 i 列后都会得到一个下三角矩阵, 行列式值为对角线上元素之积, 同时代数余子式的 $(-1)^{i+1}$ 项与子矩阵的 -1 项恰好抵消。综上所述, $G(\lambda) = \lambda^k - a_1\lambda^{k-1} - \cdots - a_k$。

因此该方法的性能瓶颈在多项式取模上, 时间复杂度 $O(k \lg k \lg n)$。引入 $\lg n$ 的原因是我们无法直接构造一个多项式 x^n 然后以 n 的规模做取模, 由于该多项式较为简单, 可以使用类似模意义快速幂的方法做模 $R(M)$ 意义下的多项式快速幂。当 k 较小时, 可以考虑暴力取模, 时间复杂度 $O(k^2 \lg n)$。

好写的优化技巧:

- 预处理出 $G(x)$ 与 $G_{rev}^{-1}(x)$ 的点值表达。
- 需要取模时才实例化取模。

参考代码:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>
const int size = 1 << 16, mod = 998244353;
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
```

```
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res ? (flag ? mod - res : res) : 0;
}
typedef long long Int64;
#define asInt64 static_cast<Int64>
int add(int a, int b) {
    a += b;
    return a < mod ? a : a - mod;
}
int sub(int a, int b) {
    a -= b;
    return a >= 0 ? a : a + mod;
}
Int64 powm(Int64 a, int k) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
int tot, root[size], invR[size];
void init(int n) {
    tot = n;
    Int64 base = powm(3, (mod - 1) / n);
    Int64 invBase = powm(base, mod - 2);
    root[0] = invR[0] = 1;
    for(int i = 1; i < n; ++i)
        root[i] = root[i - 1] * base % mod;
    for(int i = 1; i < n; ++i)
        invR[i] = invR[i - 1] * invBase % mod;
}
void NTT(int n, int* A, const int* w) {
    for(int i = 0, j = 0; i < n; ++i) {
        if(i < j)
            std::swap(A[i], A[j]);
        for(int l = n >> 1; (j ^= l) < l; l >>= 1)
            ;
    }
    for(int i = 2; i <= n; i <<= 1) {
        int m = i >> 1, fac = tot / i;
        for(int j = 0; j < n; j += i)
            for(int k = 0; k < m; ++k) {
                int &x = A[j + k], &y = A[j + k + m];
                int t = asInt64(w[k * fac]) * y % mod;
                y = sub(x, t);
                x = add(x, t);
```

```
            }
        }
    }
    typedef std::vector<int> Poly;
    void DFT(int n, Poly& A) {
        NTT(n, A.data(), root);
    }
    void IDFT(int n, Poly& A, int rn) {
        NTT(n, A.data(), invR);
        Int64 div = powm(n, mod - 2);
        for(int i = 0; i < rn; ++i)
            A[i] = A[i] * div % mod;
        memset(A.data() + rn, 0, sizeof(int) * (n - rn));
    }
    int getSize(int n) {
        int p = 1;
        while(p < n)
            p <<= 1;
        return p;
    }
    void copy(Poly& dst, const Poly& src, int siz) {
        memcpy(dst.data(), src.data(), sizeof(int) * siz);
    }
    void inv(int n, const Poly& sf, Poly& g) {
        if(n == 1)
            g[0] = powm(sf[0], mod - 2);
        else {
            int h = (n + 1) >> 1;
            inv(h, sf, g);
            int p = getSize(2 * n);
            DFT(p, g);

            Poly f(p);
            copy(f, sf, n);
            DFT(p, f);

            for(int i = 0; i < p; ++i) {
                g[i] = (2 - asInt64(g[i]) * f[i]) % mod *
                    g[i] % mod;
                if(g[i] < 0)
                    g[i] += mod;
            }
            IDFT(p, g, n);
        }
    }
    int k, p;
    Poly invGR, G;
    void polyMod(Poly& A) {
        Poly B(p);
        std::reverse_copy(A.begin() + k,
```

```
                    A.begin() + 2 * k - 1,
                    B.begin());
        DFT(p, B);
        for(int i = 0; i < p; ++i)
            B[i] = asInt64(B[i]) * invGR[i] % mod;
        IDFT(p, B, k - 1);
        std::reverse(B.begin(), B.begin() + k - 1);
        DFT(p, B);
        for(int i = 0; i < p; ++i)
            B[i] = asInt64(B[i]) * G[i] % mod;
        IDFT(p, B, k);
        for(int i = 0; i < k; ++i)
            A[i] = sub(A[i], B[i]);
        memset(A.data() + k, 0, sizeof(int) * (k - 1));
    }
    struct LazyPoly {
        Poly poly;
        int k;
        void init() {
            if(poly.empty()) {
                poly.resize(p);
                poly[k] = 1;
            }
        }
    };
    int main() {
        int n = read();
        k = read();
        p = getSize(2 * k);
        init(p);
        G.resize(p);
        G[0] = 1;
        for(int i = 1; i <= k; ++i) {
            int x = read();
            G[i] = (x ? mod - x : 0);
        }
        invGR.resize(p);
        inv(k - 1, G, invGR);
        DFT(p, invGR);
        std::reverse(G.begin(), G.begin() + k + 1);
        DFT(p, G);
        LazyPoly res, mul;
        res.k = 0, mul.k = 1;
        while(n) {
            if(n & 1) {
                if(res.poly.empty() && mul.poly.empty() &&
                   res.k + mul.k < k)
                    res.k += mul.k;
                else {
                    res.init();
```

```
                mul.init();
                DFT(p, res.poly);
                Poly tmp = mul.poly;
                DFT(p, tmp);
                for(int i = 0; i < p; ++i)
                    res.poly[i] =
                        asInt64(res.poly[i]) * tmp[i] %
                        mod;
                IDFT(p, res.poly, 2 * k - 1);
                polyMod(res.poly);
            }
        }
        n >>= 1;
        if(mul.poly.empty() && mul.k * 2 < k)
            mul.k *= 2;
        else {
            mul.init();
            DFT(p, mul.poly);
            for(int i = 0; i < p; ++i)
                mul.poly[i] = asInt64(mul.poly[i]) *
                    mul.poly[i] % mod;
            IDFT(p, mul.poly, 2 * k - 1);
            polyMod(mul.poly);
        }
    }
    res.init();
    int ans = 0;
    for(int i = 0; i < k; ++i)
        ans = (ans + asInt64(res.poly[i]) * read()) %
            mod;
    printf("%d\n", ans);
    return 0;
}
```

上述内容参考了《线性代数及其应用》[13]4.8 节以及 shadowice1984 的博客 [3]。

[3]题解 P4723【【模板】线性递推】https://www.luogu.org/blog/ShadowassIIXVIIIIV/solution-p4723

Chapter 9

数学杂项

9.1 泰勒级数展开 272
9.1.1 形式 272
9.1.2 常见泰勒级数展开 272
9.2 Simpson 积分 273
9.2.1 形式与推导 273
9.2.2 自适应 Simpson 273
9.3 概率与期望 274
9.3.1 全概率公式 274
9.3.2 贝叶斯定理 274
9.3.3 期望 274
9.3.4 伯努利试验 275
9.4 生成函数 276
9.4.1 普通型生成函数 276
9.4.2 指数型生成函数 276
9.4.3 自然数幂求和 276
9.4.4 生成函数处理背包问题 277
9.4.5 递推式与生成函数之间的转换 278
9.4.6 快速求指定数集的幂和 278
9.4.7 概率生成函数 278
9.5 拉格朗日插值 279
9.5.1 原理 279
9.5.2 插值多项式计算 279
9.5.3 多点插值 280
9.5.4 缺点 280
9.5.5 求解自然数幂和 280
9.6 反演 281
9.6.1 反演定义 281
9.6.2 二项式反演 281
9.6.3 斯特林反演 282
9.6.4 子集反演 283
9.6.5 多重子集反演 283
9.6.6 最值反演(minmax 容斥) 283

9.6.7 单位根反演 285
9.6.8 拉格朗日反演 285
9.7 常见数学公式与应用 286
9.7.1 常见公式 286
9.7.2 调和级数应用 286
9.8 线性时间复杂度整点插值 287
9.9 Berlekamp-Massey 算法 288
9.10 生成图计数 290
9.10.1 基本要点 291
9.10.2 实例 291
9.11 特征方程 292

9.1 泰勒级数展开

9.1.1 形式

函数 f 在点 a 上的泰勒级数展开为

$$\sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

$a=0$ 时称为麦克劳林级数。

9.1.2 常见泰勒级数展开

通常使用麦克劳林级数推导。

$$\begin{aligned}
e^x &= \sum_{n=0}^{\infty} \frac{x^n}{n!} \\
\ln(1-x) &= -\sum_{n=1}^{\infty} \frac{x^n}{n} \text{ for all } |x|<1 \\
\frac{1}{1-x} &= \sum_{n=0}^{\infty} x^n \\
\frac{1}{(1-x)^k} &= \sum_{n=0}^{\infty} (x^n)^{(k)} \\
(1+x)^\alpha &= \sum_{n=0}^{\infty} \binom{\alpha}{n} x^n \\
\sin(x) &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \\
\cos(x) &= \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}
\end{aligned}$$

9.2 Simpson 积分

9.2.1 形式与推导

Simpson 积分是用二次函数拟合等距三点 $(l, f(l)), (m, f(m)), (r, f(r))$ $(m = \frac{l+r}{2})$ 来积分的。推导如下：

$$\begin{aligned}\int_l^r f(x)\mathrm{d}x &\approx \int_l^r (ax^2+bx+c)\mathrm{d}x \\ &= \frac{a}{3}(r^3-l^3)+\frac{b}{2}(r^2-l^2)+c(r-l) \\ &= \frac{r-l}{6}(2a(r^2+rl+l^2)+3b(r+l)+6c) \\ &= \frac{r-l}{6}(f(l)+4f(m)+f(r))\end{aligned}$$

9.2.2 自适应 Simpson

当 Simpson 未能精确拟合函数值时，将其区间二分拟合。判断拟合程度可以将当前积分与子区间积分之和相比较。

```
typedef double FT;
const FT eps=1e-8;
FT f(FT x);
FT simpson(FT l,FT r,FT fl,FT fm,FT fr) {
    return (r-l)*(fl+4.0*fm+fr)/6.0;
}
FT SAAImpl(FT l,FT m,FT r,FT fl,FT fm,FT fr,FT sm) {
    FT lm=(l+m)*0.5,flm=f(lm),sl=simpson(l,m,fl,flm,fm);
    FT rm=(m+r)*0.5,frm=f(rm),sr=simpson(m,r,fm,frm,fr);
    FT esm=sl+sr;
    if(fabs(sm-esm)<eps)return esm;
    return SAAImpl(l,lm,m,fl,flm,fm,sl)+
    SAAImpl(m,rm,r,fm,frm,fr,sr);
}
FT SAA(FT l,FT r) {
    FT m=(l+r)*0.5;
    FT fl=f(l),fm=f(m),fr=f(r);
    return SAAImpl(l,m,r,fl,fm,fr,simpson(l,r,fl,fm,fr));
}
```

实际上 eps 也可以随着区间长度而缩放。

警告

- **最好不要在圆上做 Simpson 积分。**
- **注意初始定义域端点的位置(比如函数只在 $[0,50]$ 有较大贡献，其余区域贡献几乎为 0，若对 $[0,10000]$Simpson 积分则误差较大，考虑分析函数图像后倍增区间大小)。**

9.3 概率与期望

9.3.1 全概率公式

定理 9.1 若事件 A 可分为独立事件 $A_1, A_2, \cdots, A_n$，则

$$P(B) = \sum_{i=1}^{n} P(B|A_i)P(A_i)$$

9.3.2 贝叶斯定理

由

$$P(A \cap B) = P(B)P(A|B) = P(A)P(B|A)$$

可得贝叶斯定理

定理 9.2

$$P(A|B) = \frac{P(A)P(B|A)}{P(B)}$$

结合定理 9.1得

推论 9.3

$$P(A_i|B) = \frac{P(A_i)P(B|A_i)}{\sum_{j=1}^{n} P(B|A_j)P(A_j)}$$

9.3.3 期望

定理 9.4 (期望的线性性质) $E[X+Y] = E[X] + E[Y]$

定理 9.5 $E[aX] = aE[X]$

定理 9.6 若随机变量 X, Y 独立且期望 $E[XY]$ 有定义时，$E[XY] = E[X]E[Y]$。

有一个十分常用的转化：

$$E[X] = \sum_{i=0}^{n} i \cdot P(X = i) = \sum_{i=0}^{n} i(P(X \geq i) - P(X \geq i+1)) = \sum_{i=1}^{n} P(X \geq i)$$

定理 9.7 (Jensen's Inequality) 若函数 $f(x)$ 为凸函数（即对于任意 x, y 和 $\lambda \in [0,1]$，有 $f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)$），则 $E[f(X)] \geq f(E[X])$。

定理 9.8 对于在 $[0,1]$ 上均匀分布的随机变量 X，n 个随机变量的第 k 小值的期望为 $\frac{k}{n+1}$。

证明：随机变量 X 的概率分布函数 cdf（Cumulative Distribution Function ）为 $cdf(x) = P(X \leq x) = \int_0^1 pdf(x)\mathrm{d}x$。对其求导得到概率密度函数 pdf（Probability Density Function）：

$$pdf(x) = P(X = x) = k\binom{n}{k} x^{k-1}(1-x)^{n-k}$$

乘上随机变量后积分即为期望:

$$\int_0^1 xpdf(x)\mathrm{d}x = \frac{k}{n+1}$$

对于求解其它连续区间的期望问题也是这个思路。

9.3.3.1 随机变量的方差

$Var[X]$ 表示随机变量 X 的方差。

定理 9.9 若随机变量 X 的均值为 $E[X]$,则有 $E[X^2] = Var[X] + E^2[X]$。

定理 9.10 $Var[aX] = a^2 Var[X]$

定理 9.11 若随机变量 $X_1, X_2, \cdots, X_n$ 两两独立,则有

$$Var[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} Var[X_i]$$

9.3.3.2 高斯消元求期望

对于一般的图, 可以列出每个点期望之间的线性关系, 构造出方程组后高斯消元求解。注意要判断矩阵是否为稀疏矩阵, 若为稀疏矩阵, 继续研究其特殊性质, 一般可以不断地递推求解一元一次方程得到所有解。

对于到达目标点立即停止, 求期望步数的问题, 经典套路是将 E_u 表示为到达目标点的期望步数, 而不是表示为从起始点出发到该点的期望步数。

求**树**上期望时, 一般的套路是将方程表达为 $dp[u] = k \cdot dp[p] + b$ 的形式, 自底向上把方程推到根后再从根往下推数值解。

9.3.4 伯努利试验

每次伯努利试验有 2 种结果:以 p 的概率成功, 或者以 $q = 1 - p$ 的概率失败。

9.3.4.1 几何分布

不断进行伯努利试验, 第 k 次成功的概率为 $q^{k-1}p$。试验次数的期望为

$$\begin{aligned} E[X] &= \sum_{k=1}^{\infty} kq^{k-1}p \\ &= \frac{p}{q}\sum_{k=1}^{\infty} kq^k \\ &= \frac{p}{q} \cdot \frac{q}{(1-q)^2} (\text{参见 } 9.4) \\ &= \frac{1}{p} \end{aligned}$$

9.3.4.2 二项分布

进行 n 次伯努利试验, 成功 k 次的概率为 $\binom{n}{k} p^k q^{n-k}$。根据期望的线性性质, n 次伯努利试验的期望成功次数为 $E[X] = E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} p = np$。

以上内容参考了算法导论 [4] 附录 C.4。

9.4 生成函数

生成函数为形式幂级数, 假装其收敛。

9.4.1 普通型生成函数

对于数列 $< f_0, f_1, \cdots >$, 若形式幂级数

$$F(x) = \sum_{i=0}^{\infty} f_i x^i$$

则称 $F(x)$ 为数列 f_n 的普通型生成函数(OGF, Ordinary Generating Function)。

OGF 之积对应数列的卷积, 不考虑元素之间的顺序, 适用于组合计算。

常见 OGF:

数列	OGF
$< 1, 1, 1, \cdots >$	$\frac{1}{1-x}$
$< 0, 1, 2, 3, \cdots >$	$\frac{x}{(1-x)^2}$(将 $\frac{1}{1-x}$ 微分后平移)
$< 1. - 1.1, -1, \cdots >$	$\frac{1}{1+x}$
$< 1, c, c^2, \cdots >$	$\frac{1}{1-cx}$
$< 0, 1, \frac{1}{2}, \frac{1}{3}, \cdots >$	$\ln \frac{1}{1-x}$
$< 0, 1, 1, 2, 3, 5, \cdots >$	$\frac{x}{1-x-x^2}$

9.4.2 指数型生成函数

对于数列 $< f_0, f_1, \cdots >$, 若形式幂级数

$$F(x) = \sum_{i=0}^{\infty} f_i \cdot \frac{x^i}{i!}$$

则称 $F(x)$ 为数列 f_n 的指数型生成函数(Exponential Generating Function)。

EGF 之积对应数列卷积乘以组合数, 适合计算排列。注意 EGF 在计算时存储的是 $\frac{a_i}{i!}$, 初始化 EGF 时要将系数除以 $i!$。**启示:卷积只是优化生成函数乘法的工具,表达与计算独立。**

常见 EGF:

以下生成函数对应的数列可由泰勒展开推导。

数列	EGF
$< 1, 1, 1, \cdots >$	e^x
$< 0, 1, 2, 3, \cdots >$	xe^x
$< 1, c, c^2, \cdots >$	e^{cx}

9.4.3 自然数幂求和

记 B_n 为第 i 个伯努利数, 其中 $B_0 = 1$。首先使用公式

$$\sum_{i=1}^{n} n^k = \frac{1}{k+1} \sum_{i=1}^{k+1} \binom{k+1}{i} B_{k+1-i} (n+1)^i$$

只要预处理出逆元和伯努利数, 就可以 $O(k)$ 求出答案。

至于伯努利数, 可以使用

$$\sum_{i=0}^{n}\binom{n+1}{i}B_i=0\Leftrightarrow B_n=-\frac{1}{n+1}\sum_{i=0}^{n-1}\binom{n+1}{i}B_i$$

$O(k^2)$ 预处理。

当 k 较大时, 使用其对应的生成函数展开

$$\begin{aligned}F(x)&=\sum_{i=0}^{\infty}B_i\cdot\frac{x^i}{i!}\\&=\frac{x}{e^x-1}\\&=\frac{1}{\sum_{i=0}^{\infty}\frac{x^i}{(i+1)!}}\end{aligned}$$

$O(k\lg k)$ 多项式求逆解决。

以上内容参考了 _rqy[1]。、acdreamers[2] 和 Miskcoo[3] 的博客。

9.4.4 生成函数处理背包问题

生成函数法用来计算大规模背包的方案数。

9.4.4.1 多重背包

每个物品的生成函数为 $\sum_{k=0}^{c_i}x^{kw_i}$, 化简为 $\frac{1-x^{(c_i+1)w_i}}{1-x^{w_i}}$。由于乘积指数项的大小不可控, 考虑将乘积变为取对数求和。设答案为的生成函数为 $A(x)$, 那么有

$$\ln A(x)=\sum_{i=1}^{n}\ln(1-x^{(c_i+1)w_i})-\ln(1-x^{w_i})$$

使用泰勒展开 $\ln(1-x^k)=-\sum_{i=1}^{\infty}\frac{x^{ki}}{i}$ 得到 $lnA(x)$ 后多项式 exp 即为答案。注意 $\ln(1-x^k)$ 要合并同类项, 否则一堆 $1-x$ 可以轻易将其卡掉。合并同类项后根据调和级数可知构造多项式的复杂度为 $O(n\lg n)$, n 为背包容量。

9.4.4.2 01 背包

每个物品的生成函数为 $1+x^{w_i}$, 总方案的生成函数为这些函数之积。不过由于指数上界 $\sum w_i$ 不可控, 只能转化为多重背包解决。

9.4.4.3 完全背包

背包的容量已知, 单个物品最多装入的个数是有限的, 因此完全背包可转化为多重背包。

该讨论源自 AntiLeaf 的题目「Antileaf's Round」咱们去烧菜吧[4]。

[1]生成函数简介 | _rqy's Blog https://rqy.moe/Algorithms/generating-function/
[2]伯努利数与自然数幂和 https://blog.csdn.net/acdreamers/article/details/38929067
[3]多项式求逆元-Miskcoo's Space http://blog.miskcoo.com/2015/05/polynomial-inverse
[4]AntiLeaf's Round 题解 https://loj.ac/article/304

9.4.5 递推式与生成函数之间的转换

递推式满足 $f_n = \sum_{i=1}^{k} c_i f_{n-i}$, 记生成函数为 F。

9.4.5.1 递推式 $\rightarrow$ 生成函数

考虑生成函数的 x^n 的系数, 它的系数减去其它系数的线性加权和等于 0。为了做整个函数的加减, 使用 $c_i x_i$ 进行平移, 最后有 $F * (1 - \sum_{i=1}^{k} c_i x^i) =$ 余项的形式, 容易解出 F。

9.4.5.2 生成函数 $\rightarrow$ 递推式

由正推的过程发现分母部分为递推式的系数, 而分子部分指示了递推式的初始项。

注意如果推出了式子后为一元二次方程的形式, 不一定使用求根公式, 牛顿迭代法直接求解可以避免系数 a 无逆的情况。当然也可以通过观察把 a 的最低项消为常数。

9.4.6 快速求指定数集的幂和

该需求来自 LuoguP4705 玩游戏。

给定数集 $a_1, a_2, \cdots, a_n$, 求 $k = 1, 2, \cdots, t$, $f(k) = \sum_{i=1}^{n} a_i^k$ 的值。

考虑令 $F(x)$ 为数列 f 的生成函数,由于对于单个的 a_i,它的幂的生成函数为 $\frac{1}{1-a_i x}$,而 $F(x)$ 就是它的总和,即 $\sum_{i=1}^{n} \frac{1}{1-a_i x}$,但这个式子仍然不好算。注意到 $(\ln(1-a_i x))' = \frac{-a_i}{1-a_i x}$, 令 $G(x) = \sum_{i=1}^{n} \frac{-a_i}{1-a_i x}$, 有 $F(x) = -xG(x) + n$。将对数函数带回 $G(x)$ 并化简得 $G(x) = \left(\ln \left(\prod_{i=1}^{n} (1 - a_i x) \right) \right)'$。使用分治 NTT 很容易计算出乘积。

9.4.7 概率生成函数

例题:CTSC2006 歌唱王国

有时需要求到达目标状态的期望步数, 但例题无法使用高斯消元通过(KMP 的转移不好快速处理)。

此时可以大胆地设一个生成函数 $F(x) = \sum_{i=0}^{\infty} Pr[step = i] x^i$, 第 i 项表示结束步长为 i 的概率, 那么我们要求的期望就是 $F'(1)$。同时根据概率的性质, 有 $F(1) = 1$。

再设一个生成函数 $G(x)$, 第 i 项表示未结束步长为 i 的概率。考虑从未结束状态转移一步, 可知 $F(x) + G(x) = xG(x) + 1$。

然后再构造一个等式, 强制其结束的暴力方法就是直接加入模式串, 其生成函数就是 $G(x) \cdot (\frac{1}{n})^m$。这种方法可能导致在已结束的母串后继续加字符, 这种情况可以从 $F(x)$ 转移。考虑转移的条件, 发现模式串的末尾 i 个字符必须与开头 i 个字符匹配, 再接上 $m - i$ 个字符就能等价于强制结束的情况。记末尾 i 个字符与开头 i 个字符的匹配情况为 a_i, 其生成函数为 $\sum_{i=1}^{m} a_i \cdot F(x) \cdot (\frac{1}{n})^{m-i}$。$i \neq 0$ 是因为此类状态是从还未结束的状态开始转移的。

继续推导式子,对 $F(x)+G(x)=xG(x)+1$ 求导,可得 $F'(x)+G'(x)=xG'(x)+G(x)$,带入 $x=1$ 得 $F'(1)=G(1)$。继续带入 $x=1$ 求出 $G(1)=\sum_{i=1}^{m}a_iF(1)n^i$。又因为 $F(1)=1$,所以可以 $O(n)$ 求解 $F'(1)$(Hash/KMP 均可)。

启示:生成函数有时只是推算式子的工具,并不需要实际卷积。

该内容参考了水滴的题解[5]。

随机变量的方差也可以使用概率生成函数计算。

$$Var[X]=E[X^2]-E^2[X]=F''(1)+F'(1)-(F'(1))^2$$

该内容参考了 IOI2018 中国国家候选队论文集杨懋龙的《浅谈生成函数在掷骰子问题上的应用》。

9.5 拉格朗日插值

9.5.1 原理

拉格朗日插值法的思路是将每个点对应一个基函数,使得点 (x_i,y_i) 对应的基函数 $F_i(x)$ 满足

$$F_i(x)=\begin{cases}y_i & \text{if } x=x_i\\ 0 & \text{otherwise}\end{cases}$$

如何构造 $F_i(x)$ 呢? 首先可以提出 y_i 令 $F_i(x)=y_iG_i(x)$,然后显然 $G_i(x)$ 有 $\prod_{j\neq i}x-x_j$ 项,为了让 $G_i(x_i)=1$,再除以 $\prod_{j\neq i}x_i-x_j$。

综上所述,插值函数为

$$F(x)=\sum_{i=1}^{n}y_i\prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

9.5.2 插值多项式计算

时间复杂度 $\Theta(n^2)$,步骤如下:

1. $O(n^2)$ 计算出多项式 $F(x)=\prod_{i=1}^{i}x-x_i$;
2. $\Theta(n^2)$ 计算每个 i 的多项式 $\frac{F(x)}{x-x_i}$;
3. $\Theta(n^2)$ 将每个多项式乘以系数 $\frac{y_i}{\prod_{i\neq j}x_i-x_j}$(分母部分直接将 x_i 带入 i 对应的分子的多项式求出);
4. $\Theta(n^2)$ 将多项式相加合并。

单点插值也可以使用这种优化。

[5]P4548 [CTSC2006] 歌唱王国题解
https://www.luogu.org/problemnew/solution/P4548

9.5.3 多点插值

- 预处理出多项式, 然后每次使用霍纳法则 $O(n)$ 求值;
- 预处理出每个 i 的 $y_i \prod_{i \neq j} \frac{1}{x_i - x_j}$, 每次 $O(n^2)$ 求分子后相加。

9.5.4 缺点

- 拉格朗日插值法的给出的曲线的确经过了样本点, 但是这个曲线有可能十分曲折, 而且受单个点的影响大。可以考虑使用最小二乘逼近来消除噪声, 或者使用样条。不过拉格朗日插值一般用于已确定多项式次数的插值, 比如自然数幂和。
- $\Theta(n^2)$ 拉格朗日插值法的时间复杂度并没有比 FFT 插值方法优。

9.5.5 求解自然数幂和

易知 $F_k(n) = \sum_{i=0}^{n} i^k$ 是一个 $k+1$ 次多项式。预处理出 F_k 在 $1, \cdots, k+2$ 上的值, 插值出多项式。

在预处理时直接处理出多项式而不是点值表示, 插值的复杂度要 $O(n)$ 而不是 $O(n^2)$。

代码模板:

```
struct Poly {
    Int64 fac[maxk];
    int end;
    void init(int k) {
        int val[maxk];
        end = k + 1;
        for(int i = 0; i <= end; ++i) {
            val[i] = 1;
            for(int j = 0; j < k; ++j)
                val[i] = asInt64(val[i]) * i % mod;
        }
        for(int i = 1; i <= end; ++i)
            val[i] = (val[i - 1] + val[i]) % mod;
        Int64 kt[maxk];
        for(int i = 0; i <= end; ++i)
            fac[i] = 0;
        for(int i = 0; i <= end; ++i) {
            kt[0] = val[i];
            for(int j = 1; j <= end; ++j)
                kt[j] = 0;
            for(int j = 0; j <= end; ++j)
                if(i != j) {
                    for(int k = end; k >= 1; --k)
                        kt[k] =
                            (kt[k - 1] - j * kt[k]) %
                            mod;
                    kt[0] = -j * kt[0] % mod;
                    Int64 div = powm(i - j, mod - 2);
```

```
                    for(int k = 0; k <= end; ++k)
                        kt[k] = kt[k] * div % mod;
                }
            for(int j = 0; j <= end; ++j)
                fac[j] = (fac[j] + kt[j]) % mod;
        }
    }
    Int64 operator()(Int64 x) const {
        x%=mod;//!!!
        Int64 res = 0;
        for(int i = end; i >= 0; --i)
            res = (res * x + fac[i]) % mod;
        return res;
    }
}
```

血泪史：写 Project Euler 639 时，由于插值处没有对 x 预先取模，导致在保持系数在 $(-mod, mod)$ 与 $[0, mod)$ 的答案不一致，且它们的答案都是错误的，我因此调试了一天。

Update：注意到由于插值点是连续的，$\prod_{i\neq j} i-j$ 实际上可以表示为阶乘之积，可以 $O(n)$ 预处理，因此预处理复杂度也可以达到 $O(n)$。

9.6 反演

9.6.1 反演定义

若数列 f_n 与 g_n 满足

$$f_n = \sum_{i=0}^{n} a_{ni} g_i$$

反演就是已知数列 f_n（函数较简单），求数列 g_n 的过程（关键是要推出 b_{ni}）：

$$g_n = \sum_{i=0}^{n} b_{ni} f_i$$

这其实是一个求解线性方程组的过程。

9.6.2 二项式反演

定理 9.12

$$f(n) = \sum_{i=0}^{n} (-1)^i \binom{n}{i} g(i) \Leftrightarrow g(n) = \sum_{i=0}^{n} (-1)^i \binom{n}{i} f(i)$$

使用容斥证明：

设集合 S 中拥有性质 $P_1, P_2, \cdots, P_n$ 的集合分别为 $A_1, A_2, \cdots, A_n$，根据定理 5.4可得不具有这 n 个性质的对象的集合大小为

$$f(n) = |S| + \sum_{\emptyset \neq J \subseteq 1,2,\cdots,n} (-1)^{|J|} \left| \bigcap_{j \in J} A_j \right|$$

若集合 $A_1, A_2, \cdots, A_n$ 满足任意 i 个集合的并集大小相等, 记为 $g(i)$, 定义 $g(0) = |S|$, 有

$$f(n) = \sum_{i=0}^{n} (-1)^i \binom{n}{i} g(i)$$

同样对 $g(i)$ 使用容斥可以得到右式。

推论 9.13

$$f(n) = \sum_{i=0}^{n} \binom{n}{i} g(i) \Leftrightarrow g(n) = \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} f(i)$$

把 $g(i)$ 代入后把外面的 $(-1)^i$ 丢进去可证。

9.6.2.1 错位排序问题

求 n 个人均站错位置的方案数。

记 $f(n)$ 为 n 个人任意站的方案数, $g(n)$ 为 n 个人都站错的方案数。

显然 $f(n) = n!$ 且 $f(n) = \sum_{i=0}^{n} \binom{n}{i} g(i)$, 由推论 9.13得

$$\begin{aligned} g(n) &= \sum_{i=0}^{n} (-1)^{n-i} \binom{n}{i} i! \\ &= \sum_{i=0}^{n} (-1)^{n-i} \frac{n!}{(n-i)!} \\ &= n! \cdot \sum_{i=0}^{n} \frac{(-1)^i}{i!} \end{aligned}$$

9.6.2.2 球染色问题

求用 k 种颜色给 n 个排成一排的球染色, 满足相邻球不同色且必须用上所有颜色的方案数。

记 $f(k)$ 为使用 k 种颜色, 相邻球不同色, 不要求用上所有颜色的染色方案数, $g(k)$ 为使用 k 种颜色的方案数。

那么有 $f(k) = k(k-1)^{n-1}$ 且 $f(k) = \sum_{i=0}^{k} \binom{k}{i} g(i)$。

同理可得 $g(k) = \sum_{i=2}^{k} (-1)^{k-i} \binom{k}{i} k(k-1)^{n-1}$。

9.6.3 斯特林反演

定理 9.14

$$f(n) = \sum_{i=1}^{n} \left\{ {n \atop i} \right\} g(i) \Leftrightarrow g(n) = \sum_{i=1}^{n} (-1)^{n-i} \left[{n \atop i} \right] f(i)$$

9.6.4 子集反演

定理 9.15

$$f(S)=\sum_{T\subseteq S}(-1)^{|T|}g(T)\Leftrightarrow g(S)=\sum_{T\subseteq S}(-1)^{|T|}f(T)$$

推论 9.16

$$f(S)=\sum_{T\subseteq S}g(T)\Leftrightarrow g(S)=\sum_{T\subseteq S}(-1)^{|S|-|T|}f(T)$$

证明同 9.6.2节所述。

9.6.5 多重子集反演

一个数的质因数分解可以对应一个多重子集, 考虑莫比乌斯反演。

以上内容参考了 vfleaking 的幻灯片 [6] 和 Miskcoo 的博客[7]。

9.6.6 最值反演(minmax 容斥)

9.6.6.1 一般形式

记集合为 S, $max(S)$ 为集合 S 的最大元素, $min(S)$ 为集合 S 的最小元素。实际上这两个函数可推广为关于最值的函数。

最值反演的公式如下:

$$\begin{aligned}max(S)&=\sum_{T\subseteq S}(-1)^{|T|+1}min(T)\\min(S)&=\sum_{T\subseteq S}(-1)^{|T|+1}max(T)\end{aligned}$$

证明(以第一个式子为例): 记 $x=max(S)$(如果集合可重, 就给相同元素编号使其不相等, 下面仅考虑不可重集合的情况)。当且仅当 $T=\{x\}$ 时贡献才含有 x 项。对于 $T\neq\{x\}$, 设 $y=min(T)$, 设 S 中 $\geq y$ 的元素有 k 个, 有 $k>1$, 由于在这 k 个中选择奇数个与选择偶数个的方案数相同(使用二项式定理可证明), 最终 y 的贡献会被抵消。

9.6.6.2 期望形式

由期望的线性性质可以推出在 $max(S/T)$ 与 $min(S/T)$ 外再套上一层期望后等式仍然成立。

一般的套路是每位都有一定的概率从 0 变为 1, 求全部变为 1 的期望步数。最值反演后转换为至少 1 位变为 1 的期望步数。

例题「HAOI2015」按位或 记 $max(S)$ 为状态 S 中最晚出现的 1 的出现的期望时间, $min(S)$ 为状态 S 中最早出现的 1 的出现期望时间。$min(S)$ 很容易求得其表达式, 考虑与 S 的交不为空集的 T, 表达式为 $min(S)=\frac{1}{\sum_{T\cap S\neq\emptyset}P_T}$(根据伯努利试验中几何分布的期望求得, 参见第 9.3.4节)。这个式子仍然不好求, 可以考虑补集转换, 即考虑 T 与 S 的交为空集的情况。这个条件蕴含了 $T\subseteq C_US$, 使用第 7.4节的 FMT 可快速求出。

代码:

[6]炫酷反演魔术 http://vfleaking.blog.uoj.ac/slide/87

[7]反演魔术:反演原理及二项式反演-Miskcoo's Space.
http://blog.miskcoo.com/2015/12/inversion-magic-binomial-inversion

```cpp
#include <cctype>
#include <cstdio>
#include <cstdlib>
typedef double FT;
const FT eps = 1.0 - 1e-8;
FT readFT() {
    int c;
    do
        c = getchar();
    while(!isgraph(c));
    char str[32];
    int pos = 0;
    while(isgraph(c)) {
        str[pos++] = c;
        c = getchar();
    }
    str[pos] = '\0';
    return strtod(str, 0);
}
const int size = 1 << 20;
FT P[size];
int bc[size];
int main() {
    int n;
    scanf("%d", &n);
    int end = 1 << n;
    for(int i = 0; i < end; ++i)
        P[i] = readFT();
    for(int i = 1; i < end; i <<= 1)
        for(int j = 0; j < end; ++j)
            if(j & i)
                P[j] += P[j ^ i];
    int mask = end - 1;
    FT ans = 0.0;
    for(int i = 1; i < end; ++i) {
        bc[i] = bc[i >> 1] + (i & 1);
        FT p = P[mask ^ i];
        if(p >= eps) {
            puts("INF");
            return 0;
        }
        FT w = 1.0 / (1.0 - p);
        if(bc[i] & 1)
            ans += w;
        else
            ans -= w;
    }
    printf("%.10lf\n", ans);
    return 0;
}
```

9.6.6.3 推广

S 的第 k 大 $max_k(S)$ 满足如下等式:

$$max_k(S) = \sum_{T\subseteq S} (-1)^{|T|+k} \binom{|T|-1}{k-1} min(T)$$

证明留坑待补。
上述内容参考了 Mr_Spade 的博客[8]。

9.6.7 单位根反演

单位根反演用来解决已知生成函数 $F(x) = \sum_{i=0}^{n} f(i)x^i$, 求 $\sum_{i=0}^{n} f(i)[k|(i+b)]$, k, b 为定值。

考虑 b 为 0 的情况, 有下列定理:

定理 9.17

$$[k|i] = \frac{1}{k}\sum_{j=0}^{k-1} \omega_k^{ij}$$

证明: 与 IDFT 的证明类似, 若 $[k \nmid i]$, 则根据求和引理 7.4, 该式的值为 0。否则 ω_k^{ij} 恒为 1, 该式的值为 1。

该定理在模意义下仍然成立。~~一般来说模意义下的给定的 k 都是 2 的幂, 因为给定模数欧拉函数值要求 2 的幂次足够大~~。

接下来考虑将要求值的式子展开, 有 $\sum_{i=0}^{n} f(i) \cdot \frac{1}{k}\sum_{j=0}^{k-1} \omega_k^{ij}$, 交换求值顺序化简为 $\frac{1}{k}\sum_{i=0}^{k-1} F(\omega_k^i)$。需要用到单位根反演的场合肯定无法快速计算 F 的每一项系数, 但是容易快速求出 $F(x)$, 目前遇到的题目基本上结合二项式展开快速求出 $F(x)$ 的值。对于 $b \neq 0$ 的情况, 可以将 $F(x)$ 乘以 x^{k-b} 平移。

上述内容参考了 czyhe 的博客[9]。

9.6.8 拉格朗日反演

如果要求某个函数的反函数的某一项, 可以使用拉格朗日反演:

定理 9.18 若多项式 $F(x), G(x)$ 都没有常数项且一次项系数互为逆元, 满足 $F(G(x)) = x$, 则

$$[x_n]G(x) = \frac{1}{n}[x^{n-1}](\frac{x}{F(x)})^n$$

9.6.8.1 扩展拉格朗日反演

定理 9.19

$$[x_n]H(G(x)) = \frac{1}{n}[x^{n-1}]H'(x)(\frac{x}{F(x)})^n$$

[8] min-max 容斥/最值反演及其推广 http://www.cnblogs.com/Mr-Spade/p/9636968.html
[9] 单位根反演 https://czyhe.me/algorithm/unit-root-inv/unit-root-inv/

证明留坑待补。

这两个定理一般用在生成函数解决图的计数问题中。

上述内容参考了 ZJT 的博客[10]。

9.7 常见数学公式与应用

9.7.1 常见公式

平方和	$\sum_{k=1}^{n} k^2$	$= \frac{n(n+1)(2n+1)}{6}$
立方和	$\sum_{k=1}^{n} k^3$	$= \frac{n^2(n+1)^2}{4}$
无穷递减几何级数	$\forall \lvert x\rvert < 1, \sum_{k=0}^{\infty} x^k$	$= \frac{1}{1-x}$
调和级数	$H_n = \sum_{k=1}^{n} \frac{1}{k}$	$= \ln n + \gamma + \varepsilon_n$
Leibniz formula for π	$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$	$= \arctan 1 = \frac{\pi}{4}$
Wallis' product	$\prod_{k=1}^{\infty} \frac{2k}{2k-1} \cdot \frac{2k}{2k+1}$	$= \frac{\pi}{2}$
Basel problem	$\zeta(2) = \sum_{k=1}^{\infty} \frac{1}{k^2}$	$= \frac{\pi^2}{6}$
素数倒数和	$\sum_{p \in P, p \le n} \frac{1}{p}$	$= \ln\ln x + M + O(\frac{1}{\ln x})$ $M \approx 0.26149721$

9.7.2 调和级数应用

$\varepsilon_n \approx \frac{1}{2n}$, 欧拉-马歇罗尼常数 $\gamma \approx 0.5772156649\ldots$。

9.7.2.1 $O(n \ln n)$dp

- 两层循环分别枚举长度和关键点(相邻关键点之间的距离等于枚举长度)。
- Eratosthenes 筛法:枚举因子和因子的倍数。

9.7.2.2 书籍堆叠问题

有 n 本长度为 l, 质量分布均匀的书, 将书叠成一摞, 放在桌边, 求书最多能伸出多长。

首先书籍从下到上肯定是不断伸出的, 从上往下编号, 设第 $i-1$ 本书比第 i 本书多伸长 x_i, 记最上面的书为第 0 本书, 桌子为第 n 本书。

[10]拉格朗日反演 http://zjt-blog.cc/articles/63

若要使书不倒下,需满足上面的书的重心在当前书上,即

$$\frac{\sum_{i=1}^{k}\sum_{j=i}^{k}x_i}{k}=\frac{\sum_{i=1}^{k}ix_i}{k}\le\frac{l}{2}\ k=1,2,\cdots,n$$

当 $x_i=\frac{\frac{l}{2}}{i}$ 时等号恒成立,此时伸长量 $L=\sum_{i=1}^{n}x_i=\frac{l}{2}H_n$。
注意在计算调和级数时,小规模用暴力,大规模用 $O(1)$ 近似。
如果允许在一层上放多本书,则最大伸出量与 $\sqrt[3]{n}$ 成正比。
该问题源自 Wikipedia-EN[11]。

9.7.2.3 吉普车问题

给定 n 个单位的燃料,吉普车只能携带 1 单位的燃料,1 单位燃料可行驶 1 单位距离,吉普车可以在沙漠的任意地方留下燃料,最大化最后一次的行驶距离。
该问题有两种类型:

- 探索沙漠:最后一次要返回基地,答案为 H_n。
- 穿越沙漠:最后一次不返回基地,答案为 $\sum_{k=1}^{n}\frac{1}{2k-1}=H_{2n-1}-\frac{1}{2}H_{n-1}$。

证明留坑待补。
该问题源自 Wikipedia-EN[12]。
以上内容参考了 Wikipedia-EN[13]。

9.8 线性时间复杂度整点插值

给定 k 次多项式 $P(x)$ 在 $P(0),P(1),\cdots,P(k)$ 上的值,求 $P(n),n\in\mathbb{N}$。
考虑 $\binom{x}{i}$ 是一个关于 x 的 i 次多项式,所以可以使用 $\binom{x}{0},\binom{x}{1},\cdots,\binom{x}{k}$ 进行线性组合(因为它们的次数两两不同,线性无关),即

$$P(x)=\sum_{i=0}^{k}\binom{x}{i}p(i)\tag{9.1}$$

当 $x\le k$ 时,$P(i)$ 可以表示为

$$P(x)=\sum_{i=0}^{x}\binom{x}{i}p(i),x\le k$$

然后套推论 9.13得:

$$p(x)=\sum_{i=0}^{x}(-1)^{x-i}\binom{x}{i}P(i),x\le k\tag{9.2}$$

[11]Block-stacking problem - Wikipedia https://en.wikipedia.org/wiki/Block-stacking_problem
[12]Jeep problem - Wikipedia https://en.wikipedia.org/wiki/Jeep_problem
[13]Harmonic series (mathematics) - Wikipedia https://en.wikipedia.org/wiki/Harmonic_series_(mathematics)

将 9.2代回 9.1得:

$$\begin{aligned}
P(x) &= \sum_{i=0}^{k}\binom{x}{i}\sum_{j=0}^{i}(-1)^{i-j}\binom{i}{j}P(j)\\
&= \sum_{j=0}^{k}P(j)\sum_{i=j}^{k}(-1)^{i-j}\binom{x}{i}\binom{i}{j} \text{（提出 } P(j)\text{）}\\
&= \sum_{j=0}^{k}P(j)\sum_{i=0}^{k-j}(-1)^{i}\binom{x}{i+j}\binom{i+j}{j}\\
&= \sum_{j=0}^{k}\binom{x}{j}P(j)\sum_{i=0}^{k-j}(-1)^{i}\binom{x-j}{i}\\
&= \sum_{j=0}^{k}\binom{x}{j}P(j)\binom{k-x}{k-j} \text{（根据定理 6.11）}\\
&= \sum_{j=0}^{k}(-1)^{k-j}\binom{x}{j}\binom{x-j-1}{k-j}P(j) \text{（根据引理 6.12）}
\end{aligned}$$

$$\text{其中 } \binom{x}{j}\binom{x-j-1}{k-j} = \frac{x!}{(x-j)(x-k-1)!}\cdot\frac{1}{j!(k-j)!}$$

对组合数部分划分后,左边部分的 $\frac{x!}{(x-k-1)!}$ 是 $k+1$ 个值之积,$x-j$ 是其中的值,$O(k)$ 预处理前缀积与后缀积,$O(1)$ 合并。右边部分使用线性推逆元 $O(k)$ 完成。

以上内容参考了 Miskcoo 的博客[14]。

9.9 Berlekamp-Massey 算法

BM 算法用来求一个序列的最短线性递推式。

BM 算法维护当前的线性递推式, 按顺序将元素加入序列, 若不满足当前线性递推式则根据历史信息矫正线性递推式。

具体算法如下(使用生成函数 $A(x)$ 来表示递推式, 下标从 1 开始, 常数项表示当前项的系数, 最后要被缩放到-1):

1. 维护当前递推式 $A(x)$, 上次失配时的递推式 $B(x)$, 以及上次失配时 $B(x)$ 的值与失配位置值的差值 ld, 当前位置与上次失配位置的偏移 off。
2. 计算 $A(x)$ 在当前位置的值:
   - 若与序列的值相等, 则迭代检查下一项。
   - 否则记当前差值为 cd, 矫正多项式为 $A'(x) = A(x) - \frac{cd}{ld}x^{off}B(x)$。

接下来证明算法正确性:

- 当前位置满足递推式 $A'(x)$: 注意到 x^{off} 项将 $B(x)$ 的取值移动到了上次失配位置, 差值为 ld, 而递推式 $A(x)$ 的差值为 cd, 所以 $A'(x)$ 的差值被消为 0。

[14]特殊多项式在整点上的线性插值方法-Miskcoo's Space
http://blog.miskcoo.com/2015/08/special-polynomial-linear-interpolation

- 先前位置满足递推式 $A'(x)$:首先所有位置满足 $A(x)$;因为 $[1,$ 上次失配位置$)$ 的位置满足递推式 $B(x)$,所以 $[off+1\ off+$ 上次失配位置$)$ 的位置满足递推式 $x^{off}B(x)$;而 $[1,off]$ 的位置全部作为递推式 $x^{off}B(x)$ 的递推基础,也满足递推式。

注意由于 ld 仅在失配时改变,必定存在乘法逆元(实现时仅存储其乘法逆元,然后从 0 阶递推式开始,初始 ld 任选一个非 0 值)。

代码如下:

```
#include <algorithm>
#include <cstdio>
#include <vector>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 3005, mod = 1000000007;
typedef long long Int64;
#define asInt64 static_cast<Int64>
Int64 inv(Int64 a) {
    Int64 res = 1;
    int k = mod - 2;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
std::vector<int> A, B;
int S[size];
int main() {
    freopen("BM-in.txt", "r", stdin);
    int n = read();
    int ld = 1, lp = 0;
    A.push_back(mod - 1);
    B.push_back(mod - 1);
    for(int i = 1; i <= n; ++i) {
        S[i] = read();
        int val = 0;
        for(int j = 0; j < A.size(); ++j)
            val =
                (val + asInt64(A[j]) * S[i - j]) % mod;
        if(val) {
            int off = i - lp;
```

```
            int cd =
                std::max(A.size(), B.size() + off);
            B.resize(cd);
            Int64 fac = asInt64(val) * ld % mod;
            if(fac)
                fac = mod - fac;
            for(int j = cd - 1; j >= 0; --j) {
                int av = (j < A.size() ? A[j] : 0);
                int bv = (j >= off ? B[j - off] : 0);
                B[j] = (av + fac * bv) % mod;
            }
            A.swap(B);
            while(A.back() == 0)
                A.pop_back();
            lp = i, ld = inv(val);
        }
    }
    for(int i = A.size(); i <= n; ++i) {
        int val = 0;
        for(int j = 0; j < A.size(); ++j)
            val =
                (val + asInt64(A[j]) * S[i - j]) % mod;
        if(val != 0)
            throw;
    }
    printf("%d\n", A.size() - 1);
    Int64 k = asInt64(mod - 1) * inv(A[0]) % mod;
    for(int i = 0; i < A.size(); ++i)
        A[i] = A[i] * k % mod;
    for(int i = 1; i < A.size(); ++i)
        printf("%d ", A[i]);
    return 0;
}
```

算法最坏时间复杂度 $O(n^2)$, 但该算法似乎只能保证给出可行解, 是否能给出最短线性递推式有待考证。

BM 算法的应用还不太多, 一般用来求 DP 值的线性递推式, 然后搭配线性递推快速求出第 n 项的值。

上述内容参考了 Rayment_cc 的博客[15]。

测试数据来自 fjzzq2002 的博客[16]。

9.10 生成图计数

鉴于生成函数在生成图计数中的应用广泛且逐渐普及, 另开一节记录生成图计数的要点。

[15] Berlekamp–Massey 算法简要介绍
https://blog.csdn.net/As_A_Kid/article/details/86286891

[16] Berlekamp-Massey 算法简单介绍
https://www.cnblogs.com/zzqsblog/p/6877339.html

下面的内容参考了 WC2019 汪乐平的讲义《生成函数，多项式算法与图的计数》、WC2017 营员交流汪乐平的课件《仙人掌计数》以及 2015 年国家集训队论文集中金策的《生成函数的运算与组合计数问题》。当时我在现场冬眠

9.10.1 基本要点

9.10.1.1 各类生成函数的用途

- 普通生成函数通常用在无标号的计数中。
- 指数生成函数通常用在有标号的计数中。
- 特殊生成函数 (SGF) 的表达式如下：

$$G(x)=\sum_{n=0}^{\infty}\frac{a_n}{2^{\binom{n}{2}}n!}x^n$$

一般用于有标号有向图的计数。$2^{\binom{n}{2}}$ 用于确定任意两点之间是否连边。

9.10.1.2 是否连通?

无标号集合计数 假设我们知道了元素集合组成的 OGF 为 $A(x)$, 那么 n 个元素组合对应的 OGF 为 $A(x)^n$, 答案即为 $\sum_{n=0}^{\infty}A(x)^n=\frac{1}{1-A(x)}$。使用多项式求逆解决。

有标号集合计数 假设我们知道了元素集合组成的 EGF 为 $A(x)$, 那么 n 个元素排列对应的 EGF 为 $A(x)^n$, 由于集合内无顺序关系, 要再除以 $n!$。答案即为 $\sum_{n=0}^{\infty}\frac{A(x)^n}{n!}=e^{A(x)}$。使用多项式 exp/ln 解决。

实际问题转化 通常可以将元素视为单个连通图, 而将集合视为多个连通图的组合。

9.10.1.3 有无标号?

有标号与无标号的方案数只差 $n!$。如果仅仅是有根与无根的区别, 系数差 n。一般先计算有标号/根的方案数, 再推回无标号/根。

9.10.1.4 复合逆

有些题目仅仅要求生成函数的某一项, 有时会推出符合逆的式子, 使用 9.6.8所述的拉格朗日反演解决。

9.10.2 实例

9.10.2.1 有标号森林

因为森林是由多棵树组成的, 因此先考虑有标号树的 EGF。
这是一个完全图生成树问题, 有两种解法：

- 根据 13.4所述的矩阵树定理计算行列式。

- 根据 11.5所述的 Purfer 序列可知, 一个 Purfer 序列唯一对应一棵有标号树, 序列长度为 $n-2$, 因此方案数为 n^{n-2}。

  那么有标号树的 EGF 为 $G(x)=\sum_{n=0}^{\infty}\frac{n^{n-2}}{n!}x^n$。

  将森林视为树的集合, 有标号森林的 EGF 为 $F(x)=e^{G(x)}$。

9.10.2.2 有标号无向连通图

多个有标号无向连通图可组合为有标号无向简单图, 考虑有标号无向简单图 EGF 的计算。

枚举完全图的每条边是否选中, 大小为 n 的无向简单图个数为 $2^{\binom{n}{2}}$。那么 EGF 为 $G(x)=\sum_{n=0}^{\infty}\frac{2^{\binom{n}{2}}}{n!}x^n$。

将简单图视为连通图的集合, 记有标号无向连通图的 EGF 为 $F(x)$, 有 $G(x)=e^{F(x)}$, 解得 $F(x)=\ln G(x)$。

9.10.2.3 有标号仙人掌

无根仙人掌不太好去重, 考虑计算有根仙人掌的个数, 根或者仙人掌不同都视为不同的有根仙人掌。记有根仙人掌的 EGF 为 $G(x)$, 无根仙人掌的 EGF 为 $F(x)=\int\frac{G(x)}{x}\mathrm{d}x$ (~~说白了就是对应系数除以 n~~)。

接下来考虑有根仙人掌的计数。删掉从根出发的边, 讨论每条边是否在环上, 以及对应子连通块的 EGF。

- 边不在环上: 删掉这条边后, 分离出的连通块也是一棵仙人掌, 以边的另一个端点为子仙人掌的根, 对应了一棵新的仙人掌, EGF 为 $G(x)$。
- 边在环上: 显然要删去 2 条边才能分离出子连通块, 将其一起考虑。设环上删去根节点后有 i 个点($i\geq 2$), 以每个点为根都有一棵仙人掌(不往环上的点走), 这些仙人掌串成了一条链, 由于链可以翻转, EGF 为 $\frac{G(x)^i}{2}$。

考虑删去这条边后子连通块的 EGF, 即上面分析出的所有情况相加, 有

$$G(x)+\sum_{i=2}^{\infty}\frac{G(x)^i}{2}=\frac{2G(x)-G^2(x)}{2-2G(x)}$$

将子连通块组合并加回根节点, 有 $G(x)=xe^{\frac{2G(x)-G^2(x)}{2-2G(x)}}$。$G(x)$ 可以使用牛顿迭代法计算。

9.10.2.4 有标号有向无环图

9.10.2.5 有标号环

9.10.2.6 无标号环

9.10.2.7 带边数限制的有标号无向连通图

9.10.2.8 有标号点双连通分量

9.11 特征方程

特征方程用于推导线性递推数列的通项。

给定数列递推式 $a_n = \sum_{i=1}^{k} c_i a_{n-i}$, 以及数列初始项 $a_0, a_1, \cdots, a_{k-1}$, 求 a_n。

虽然求 a_n 可以使用矩阵快速幂或者常系数项齐次线性递推解决, 但是像「JLOI2015」有意义的字符串这类涉及取整的题目, 它们毫无办法。因此研究 a_n 的通项并且利用其中的性质十分重要。

首先将递推式改写为特征方程 $x^k = \sum_{i=1}^{k} c_i x^{k-i}$, 然后求出它的根 $x_1, x_2, \cdots, x_k$, 那么 a_n 就是这些根 n 次幂的线性加权和。带入前 k 项求出系数就可以得到通项。

注意以下特殊情况:

- 重根:若根 x 重复 b 次, 则其系数为 $\sum_{i=0}^{b-1} f_i n^i$ 的形式。
- 复根:若 r 为复根, 将 r 表示为 $se^{\omega i}$ 的形式, 将 r^k 拆为 $s^k \cos k\omega + is^k \sin k\omega$, 实部和虚部都是合法的实根。

Chapter 10

动态规划

10.1 背包优化 295
10.1.1 bitset 优化 295
10.1.2 完全背包优化 295
10.1.3 多重背包优化 295
10.2 数位动规 297
10.3 基于连通性状态压缩的动态规划/轮廓线 DP/插头 DP 297
10.3.1 简单回路问题 297
10.3.2 简单路径问题 305
10.3.3 最大化回路/路径/连通块点权和 305
10.3.4 广义括号表示法 306
10.3.5 棋盘染色问题 306
10.3.6 局部连通性加速 DP 306
10.4 单调队列优化 306
10.5 斜率优化 307
10.5.1 推导 307
10.5.2 应用 307
10.5.3 树上斜率优化 308
10.5.4 CDQ 分治维护凸包 309
10.6 四边形不等式优化 309
10.7 矩阵快速幂优化 310
10.7.1 常规矩阵快速幂 310
10.7.2 矩阵链乘秩分解 313
10.7.3 dp 步伐不一致时的解决方案 316
10.7.4 矩阵对角化加速快速幂 316
10.7.5 固定转移矩阵多询问 DP 317
10.8 动态 dp 317
10.8.1 常规树上 DDP 方法 317
10.8.2 全局平衡二叉树 322
10.8.3 子树 DP 值查询 326
10.8.4 固定思路 327
10.9 决策单调性 DP 327
10.9.1 双指针优化 327

10.9.2 决策二分栈 DP 327
10.9.3 分治 DP 329
10.9.4 时间复杂度陷阱 334
10.9.5 决策单调性快速判断 334
10.10WQS 二分凸优化 334
10.11特殊 DP 335
10.11.1 LCIS 335
10.11.2 Dilworth 定理 336
10.11.3 杨氏图表(排序矩阵) 336
10.11.4 GarsiaWachs 算法 336
10.11.5 双单调性优化 337
10.11.6 折半状压 DP 337
10.12杂记 338

10.1 背包优化

10.1.1 bitset 优化

有些简单的背包问题可以使用位运算转移, std::bitset 可以减小常数。

10.1.2 完全背包优化

10.1.2.1 排序筛选

若一种物品比另一种物品代价更大, 收益更低, 直接排除。

10.1.3 多重背包优化

10.1.3.1 完全背包转换

若该物品的数量已经超过最大装载量, 直接将其当做完全背包, 单种物品 $O(V)$ 转移。

10.1.3.2 二进制优化

将数量按 $2^0, 2^1, 2^2, \cdots, 2^k, rem$ 拆分为多个物品, 然后做 01 背包, 单种物品 $O(V \lg c)$ 转移。

10.1.3.3 单调队列优化

朴素的多重背包状态转移方程为:

$$\begin{aligned} end &= min(c[i], j/v[i]) \\ dp[i][j] &= max(dp[i-1][j-k*v[i]] + k*w[i]), 0 \le k \le end \end{aligned}$$

令 $a = j/v[i], b = j\%v[i]$, 将方程转换为:

$$dp[i][b + a*v[i]] = max(dp[i-1][b + k*v[i]] - k*w[i]) + a*w[i], a - end \le k \le a$$

此时 k 表示比 a 少取 k 件。

此时 max 部分只与 k 的取值有关, 可以使用单调队列优化。每次转移时, 枚举 b, 对 k 做单调队列, 根据转移区间的移动把队首弹出队列。单种物品 $O(V)$ 转移。

该方法参考了 soloier 的博客[1]。

10.1.3.4 例题

Luogu P1776 宝物筛选 _NOI 导刊 2010 提高(02)[2]

使用了上述的完全背包转换和单调队列优化。

Luogu 1776

```
#include <algorithm>
#include <cstdio>
const int size = 40005;
int dp[size];
typedef std::pair<int, int> Info;
Info q[size];
void update(int V, int w, int v, int c) {
    for(int b = 0; b < v; ++b) {
        Info *beg = q, *end = q;
        for(int i = b, a = 0; i <= V; i += v, ++a) {
            while(beg < end && a - beg->first > c)
                ++beg;
            int delta = a * w;
            Info val =
                std::make_pair(a, dp[i] - delta);
            while(beg < end &&
                  (end - 1)->second <= val.second)
                --end;
            *(end++) = val;
            dp[i] = beg->second + delta;
        }
    }
}
int main() {
    int n, V;
    scanf("%d%d", &n, &V);
    for(int i = 0; i < n; ++i) {
        int w, v, c;
        scanf("%d%d%d", &w, &v, &c);
        if(c != 1) {
            if((c + 1LL) * v <= V) {
                update(V, w, v, c);
            } else {
                for(int i = v; i <= V; ++i)
                    dp[i] =
                        std::max(dp[i], dp[i - v] + w);
            }
```

[1]单调队列优化多重背包
https://blog.csdn.net/sinat_34943123/article/details/52857327

[2]【P1776】宝物筛选 _NOI 导刊 2010 提高(02)- 洛谷 https://www.luogu.org/problemnew/show/P1776

```
        } else {
            for(int i = V; i >= v; --i)
                dp[i] = std::max(dp[i], dp[i - v] + w);
        }
    }
    printf("%d\n", dp[V]);
    return 0;
}
```

10.2 数位动规

问题一般是询问区间内数位满足指定要求的数的个数。区间计数可转换为前缀和差分，因此原问题可转换为询问小于 n 的满足指定要求的数的个数。

一般思路如下：

1. 将 n 拆位为 $n_k, n_{k-1}, \cdots, n_1$；
2. 从最低位开始 dp 一直做到最高位；
3. 从最低位开始统计到次高位，因为这部分答案是满的；
4. 从最高位开始做到最低位，假设做到第 i 位，表示处理该位以前的位都固定为 n，枚举当前位 $j < n_i$，使用预处理的 dp 值统计答案。

优化 若固定的位不满足要求（比如对相邻位有限制）则直接退出。因为后续 dp 的数字都是不合法方案，可以忽略。

10.3 基于连通性状态压缩的动态规划/轮廓线 DP/插头 DP

10.3.1 简单回路问题

例题：ural 1519

给定一个棋盘，某些格子不能经过，其余格子必须经过，求有多少条简单回路。

10.3.1.1 最小表示法

最小表示法用于表示行内格子的连通性。有两种方法：

- 有障碍的格子标记为 0，连通的格子标记为同一个数，并且 i 比 $i+1$ 更早出现。预处理连通状态时使用 DFS 计算，同构于集合划分问题，状态总量为贝尔数。
- 有障碍的格子标记为 0，其余格子标记为与其连通的最左格子的列号。

下文均使用第一种方法。存储时将其编码为 k 进制数，$k \geq$ 连通块个数 $+1$。为了提高运算效率，将 k 对齐至 2 的幂。

10.3.1.2 状态的表示

在例题中, 每个格子与 4 个相邻格子都有可能连通, 连接的部分称为 ‘‘插头’’。由于转移是逐行进行的, 下一行的插头受到上一行下插头的影响, 因此每行需要用二进制表示当前行对应位置是否有下插头。由于最后需要围成一个回路, 还要维护行内格子之间的连通性。因此使用 $F(i,j,k)$ 表示前 i 行, 下插头状态为 j, 连通性为 k 的方案数。注意到若该格子没有下插头, 则它的连通性不影响下一行的插头。那么干脆直接记录下插头的连通性, 若不存在下插头则标记为 0。

鉴于逐格递推比逐行递推更有优势, 这里仅记录逐格递推。

用状态 $F(i,j,k)$ 表示当前处理到第 i 行, 已经处理完前 $i-1$ 行和第 i 行前 j 列的格子, 插头连通性为 k 的方案数。已决策的格子与未决策格子之间的分界线称为 ‘‘轮廓线’’。在转移的过程中, 除了 n 个下插头外, 第 i 格到第 $i+1$ 格还有一个右插头。k 从左到右表示 $n+1$ 个插头的连通性。

考虑 k 需要使用的进制, 由于一个回路最多有 $m/2$ 个连通块, 在例题中至少要用 7 进制, 使用 8 进制更加快速。

10.3.1.3 状态的转移

一般转移 逐格递推, 可以发现每移动一格最多有 2 个插头被改动。

枚举当前格子的插头, 状态共有 3 种转移:

下文的 ‘‘相接位置’’ 指代轮廓线与当前格子的两条邻接边, ‘‘对应位’’ 指转移前相接位置的位与转移后新边的位。

- 新建连通分量: 相接位置没有右插头和下插头, 当前格子有右插头和下插头, 将对应位置置为新的标号, 然后重新 $O(n)$ 扫描以保持最小表示法。
- 合并连通分量: 当前格子有左插头和上插头。若对应的右插头和下插头未连通, 则将含有这两个标号的所有位置标记为同一标号, 并重新扫描, 再将对应位置 0。若其已连通, 则只允许在最后一个非障碍格子中合并为一条回路。
- 保持连通分量: 相接位置有右插头或下插头恰好一个, 当前格子有上或左插头恰好一个, 有下或右插头恰好一个。不需要重新扫描, 可以 $O(1)$ 转移。

障碍格子处理 当遇到障碍格子时, 相接位置必须没有插头, 由于该格子不能铺线, 对应位置上没有插头, 仍然置为 0, 所以状态不变。

跨行处理 当从上一行的最后一个格子转移到下一行的第一个格子时, 不能从有右插头的状态转移。实现时保证上一行的最后一个格子不能产生右插头, 转移到首格时移位处理(将高位的竖线删除, 再添加新的无插头竖线到低位, 实际操作只要左移一个位置。**如果更高位还有存储数据, 要注意保留。**)。

始末状态 初始状态和终止状态都没有插头, 值为 0。

小结 在推导状态转移时需要注意以下方面:

- 状态表示是否存储足够的信息
- 连通分量的三种变化情况
- 移动格子时要保证所有裸露的插头都在轮廓线上
- 转移的目标状态表示的是**轮廓线上插头**的连通性。

- 转移完状态后是否需要重新扫描以保持最小表示法的性质
- 障碍格子和行首尾格子的处理
- 始末状态
- 从可行性与最优性对无效状态进行剪枝

10.3.1.4 程序的实现与优化

直接枚举状态会产生大量的无效状态, 因此使用队列从初始状态开始转移。枚举每个格子, 枚举循环队列中的状态, 计算出可以转移的目标状态。使用 Hash 表存储当前已经转移过的状态的 dp 值, Hash 表的 size 不必太大, 每转移一个格子开一个新的 Hash 表。

参考代码:

```
#include <cstdio>
#include <cstring>
#include <vector>
typedef long long Int64;
const int modu = 23003, modv = 22993;
struct HashTable {
    std::vector<std::pair<Int64, Int64> > LUT;
    std::vector<Int64> st;
    HashTable() : LUT(modu) {
        for(int i = 0; i < modu; ++i)
            LUT[i].first = -1;
    }
    void swap(HashTable& rhs) {
        LUT.swap(rhs.LUT);
        st.swap(rhs.st);
    }
    Int64& operator[](Int64 x) {
        int ha = x % modu, hb = 1 + x % modv, cur = ha;
        while(true) {
            if(LUT[cur].first == -1) {
                LUT[cur].first = x;
                st.push_back(x);
            }
            if(LUT[cur].first == x)
                return LUT[cur].second;
            cur += hb;
            if(cur >= modu)
                cur -= modu;
        }
    }
};
Int64 rescan(int m, Int64 x) {
    int remap[8], cnt = 0;
    remap[0] = 0;
    for(int i = 1; i < 8; ++i)
        remap[i] = -1;
    Int64 res = 0;
```

```
    for(int i = 0; i <= m; ++i, x >>= 3) {
        Int64 b = x & 7;
        if(remap[b] == -1)
            remap[b] = ++cnt;
        b = remap[b];
        res |= b << (i * 3);
    }
    return res;
}
int get(Int64 x, int i) {
    return (x >> (i * 3)) & 7;
}
Int64 set(Int64 x, int i, Int64 sb, Int64 db) {
    return x ^ ((sb ^ db) << (i * 3));
}
const int nlb = 7;
Int64 merge(int m, Int64 x, int a, int b, int ea,
            int eb) {
    Int64 res = 0;
    for(int i = 0; i <= m; ++i, x >>= 3) {
        Int64 cb = x & 7;
        cb = ((cb == a || cb == b) ? nlb : cb);
        if(i == ea || i == eb)
            cb = 0;
        res |= cb << (i * 3);
    }
    return rescan(m, res);
}
char A[15][15];
void tran(int i, int j, int n, int m, HashTable& lut,
          HashTable& clut, bool last) {
    bool trans = A[i][j] == '.';
    for(int k = 0; k < lut.st.size(); ++k) {
        Int64 cur = lut.st[k];
        Int64 val = lut[cur];
        if(j == 1)
            cur <<= 3;
        int la = get(cur, j - 1), lb = get(cur, j);
        int vaild = (la ? 1 : 0) + (lb ? 1 : 0);
        if(trans) {
            switch(vaild) {
                // rb
                case 0: {
                    if(i != n && j != m) {
                        Int64 dst = rescan(
                            m, set(set(cur, j - 1, 0,
                                       nlb),
                                   j, 0, nlb));
                        clut[dst] += val;
                    }
```

```
                    } break;
                    // l/t+r/b
                    case 1: {
                        int src = la | lb;
                        // b
                        if(i != n) {
                            Int64 dst = set(
                                set(cur, j - 1, la, src),
                                j, lb, 0);
                            clut[dst] += val;
                        }
                        // r
                        if(j != m) {
                            Int64 dst =
                                set(set(cur, j - 1, la, 0),
                                    j, lb, src);
                            clut[dst] += val;
                        }
                    } break;
                    // lt
                    case 2: {
                        if((la != lb) || last) {
                            Int64 dst = merge(
                                m, cur, la, lb, j - 1, j);
                            clut[dst] += val;
                        }
                    } break;
                }
            } else if(vaild == 0) {
                Int64 dst = cur;
                clut[dst] += val;
            }
        }
    }
    int main() {
        int n, m;
        scanf("%d%d", &n, &m);
        int li, lj;
        for(int i = 1; i <= n; ++i) {
            scanf("%s", A[i] + 1);
            for(int j = 1; j <= m; ++j)
                if(A[i][j] == '.')
                    li = i, lj = j;
        }
        HashTable lut;
        lut[0] = 1;
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j <= m; ++j) {
                HashTable clut;
                tran(i, j, n, m, lut, clut,
```

```
                    i == li && j == lj);
                clut.swap(lut);
            }
        printf("%lld\n", lut[0]);
        return 0;
    }
```

10.3.1.5 简单回路问题与括号表示法

事实上对于简单回路问题, 裸露的插头必须在轮廓线上, 轮廓线上的插头必定两两匹配。又因为求的是简单回路, 匹配的插头不会交叉。那么可以使用一个括号序列来表示连通性, 0 表示没有插头, 1 表示左端, 2 表示右端, 使用 4 进制表示状态。

再次根据连通分量的变化讨论转移:

- 新建连通分量:要求相接位置没有插头, 转移时将对应位分别置为 1 和 2。
- 合并连通分量: 需要按照相接位置的插头是左括号还是右括号讨论, 记右插头为 A, 下插头为 B:
  - A 左 B 左:将 B 对应的右括号修改为左括号
  - A 左 B 右:此时将连成一条回路, 当其为最后一个无障碍格子时才转移
  - A 右 B 左:不需要额外修改
  - A 右 B 右:将 A 对应的左括号修改为右括号

  最后将对应位都置为 0。
- 保持连通分量:直接继承有插头位置的左右标号

此法思维难度低, 实现简单, 程序速度快。但是括号表示法的局限性较大, 参见下文的广义表示法。

参考代码:

```
#include <cstdio>
#include <cstring>
#include <vector>
typedef long long Int64;
const int modu = 23003, modv = 22993;
struct HashTable {
    std::vector<std::pair<int, Int64> > LUT;
    std::vector<int> st;
    HashTable() : LUT(modu) {
        for(int i = 0; i < modu; ++i)
            LUT[i].first = -1;
    }
    void swap(HashTable& rhs) {
        LUT.swap(rhs.LUT);
        st.swap(rhs.st);
    }
    Int64& operator[](int x) {
        int ha = x % modu, hb = 1 + x % modv, cur = ha;
        while(true) {
```

```
            if(LUT[cur].first == -1) {
                LUT[cur].first = x;
                st.push_back(x);
            }
            if(LUT[cur].first == x)
                return LUT[cur].second;
            cur += hb;
            if(cur >= modu)
                cur -= modu;
        }
    }
};
int get(int x, int i) {
    return (x >> (i * 2)) & 3;
}
int set(int x, int i, int sb, int db) {
    return x ^ ((sb ^ db) << (i * 2));
}
int prev(int x, int i) {
    int sum = 0;
    while(true) {
        int c = get(x, i);
        if(c == 2)
            ++sum;
        else if(c == 1)
            --sum;
        if(sum == 0)
            return i;
        --i;
    }
}
int next(int x, int i) {
    int sum = 0;
    while(true) {
        int c = get(x, i);
        if(c == 1)
            ++sum;
        else if(c == 2)
            --sum;
        if(sum == 0)
            return i;
        ++i;
    }
}
char A[15][15];
void tran(int i, int j, int n, int m, HashTable& lut,
          HashTable& clut, bool last) {
    bool trans = A[i][j] == '.';
    for(int k = 0; k < lut.st.size(); ++k) {
        int cur = lut.st[k];
```

```
        Int64 val = lut[cur];
70      if(j == 1)
            cur <<= 2;
        int la = get(cur, j - 1), lb = get(cur, j);
        int vaild = (la ? 1 : 0) + (lb ? 1 : 0);
        if(trans) {
            switch(vaild) {
                // rb
                case 0: {
                    if(i != n && j != m) {
                        int dst =
80                          set(set(cur, j - 1, 0, 1),
                                j, 0, 2);
                        clut[dst] += val;
                    }
                } break;
                // l/t+r/b
                case 1: {
                    int src = la | lb;
                    // b
                    if(i != n) {
90                      int dst = set(
                            set(cur, j - 1, la, src),
                            j, lb, 0);
                        clut[dst] += val;
                    }
                    // r
                    if(j != m) {
                        int dst =
                            set(set(cur, j - 1, la, 0),
                                j, lb, src);
100                     clut[dst] += val;
                    }
                } break;
                // lt
                case 2: {
                    int dst = cur;
                    if(la == 1) {
                        if(lb == 1) {
                            int pos = next(cur, j);
                            dst = set(cur, pos, 2, 1);
110                     } else if(!last)
                            continue;
                    } else if(lb == 2) {
                        int pos = prev(cur, j - 1);
                        dst = set(cur, pos, 1, 2);
                    }
                    dst = set(set(dst, j - 1, la, 0),
                              j, lb, 0);
                    clut[dst] += val;
```

```
                } break;
            }
        } else if(vaild == 0) {
            int dst = cur;
            clut[dst] += val;
        }
    }
}
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    int li, lj;
    for(int i = 1; i <= n; ++i) {
        scanf("%s", A[i] + 1);
        for(int j = 1; j <= m; ++j)
            if(A[i][j] == '.')
                li = i, lj = j;
    }
    HashTable lut;
    lut[0] = 1;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j) {
            HashTable clut;
            tran(i, j, n, m, lut, clut,
                 i == li && j == lj);
            clut.swap(lut);
        }
    printf("%lld\n", lut[0]);
    return 0;
}
```

10.3.1.6 非回路问题转化为简单回路问题

求从棋盘中一个特殊点经过所有非障碍点走到另一个特殊点的方案数。

可以尝试额外构造一条宽度为 1 的路径使其连通, 新棋盘的一条回路对应了原棋盘的一条路径。

10.3.2 简单路径问题

给定一个棋盘, 某些格子不能经过, 其余格子必须经过, 求有多少条简单路径。

此时非轮廓线上有不超过 2 个裸露插头。若使用最小表示法, 则还要记录每个插头与路径端点的连通情况。若使用括号表示法, 则再引入标号 3 指示独立插头, 说明这个插头连接着路径的一端, 转移时需要保证任意时刻轮廓线上的独立插头不超过 2 个。

10.3.3 最大化回路/路径/连通块点权和

- 回路: 注意仅在连为回路的情况更新答案。由于允许有些格子不经过, 会导致回路外还可能存在一些孤立的路径, 需要特判连为回路后轮廓线上是否均没有插头。

- 路径:额外使用 2bit 记录轮廓线上方裸露插头的个数,同时引入独立插头标记。注意要始终保证裸露插头的个数 ≤ 2。读入时要顺便计算只选取一个格子的方案。
- 连通块:仅记录 m 个格子的连通性。特别考虑移动轮廓线后消失的格子(即当前格子的上方),若它被选中且当前格子不选,需要保证它至少与轮廓线上的其它格子连通。

由于要求至少选择一个格子/回路,要在计算的过程中更新答案。必要时单独计算选取一个格子的特殊情况。已经形成回路/路径的方案,不必加入下一次迭代。

10.3.4 广义括号表示法

括号表示法的局限性在于其只能表示在轮廓线上最多有 2 个插头的连通分量。

将最左插头标记为 "(",最右插头标记为 ")",中间的插头标记为 ")(",独立插头标记为 "()",能够匹配的括号对应的插头是连通的。

10.3.5 棋盘染色问题

棋盘上格子的连通性取决于棋盘格子的颜色,因此需要额外记录轮廓线上棋盘格子的颜色。

10.3.6 局部连通性加速 DP

有些题目会给出一个特殊的构图方法,其连边具有局部性质。那么就可以讨论局部连通性计算转移矩阵,使用矩阵快速幂加速 DP。

上述内容参考了 IOI2018 国家集训队论文集中陈丹琦的《基于连通性状态压缩的动态规划问题》。

10.4 单调队列优化

当状态转移方程为如下形式时

$$dp[i][j] = max(dp[i-1][j-k] + w(i-1, j-k)) + c(i,j), l \leq k \leq r$$

可以使用单调队列优化。

该状态转移方程满足如下性质:

- 转移区间是连续的。
- 当前转移位置到转移区间有距离限制。

对每个 i 进行 dp 的步骤如下:

1. 初始化空队列;
2. 计算自己到队首的距离,弹出超出转移距离的队首;
3. 计算新进入转移区间的转移点的权(即 max 的内容);
4. 为了维护队列的单调性,不断地从队尾弹出不比当前转移点更优的旧转移点(即使同样优也要弹出,因为当前转移点肯定比旧转移点更晚弹出);
5. 此时队首为最优值,加上常数后即为 dp 最优值。

10.5 斜率优化

10.5.1 推导

当状态转移方程为如下形式时：

$$dp[i] = min(a[i] * b[j] + c[i] + d[j])$$

考虑使用斜率优化。

以下推导假设 $a[i]$ 单调递减且 $b[j]$ 单调递增：

设决策点 $j < k < i$，且从点 k 转移到 i 不差于从点 j 转移到 i，易证从点 k 转移到 $i+1$ 同样不差于从点 j 转移到 $i+1$。称该性质为决策单调性。

接下来考虑点 k 不比点 j 更差的条件：

$$\begin{aligned} a[i] * b[j] + c[i] + d[j] &\geq a[i] * b[k] + c[i] + d[k] \\ \Rightarrow -a[i] &\geq \frac{d[k] - d[j]}{b[k] - b[j]} \end{aligned}$$

记斜率

$$slope(i, j) = \frac{d[j] - d[i]}{b[j] - b[i]}$$

斜率可以使用单调队列维护，记 $q[b]$ 为队首，$q[e-1]$ 为队尾：

- 若 $-a[i] \geq slope(q[b], q[b+1])$，则表明 $q[b+1]$ 不比 $q[b]$ 更差，弹出 $q[b]$。
- 若 $slope(q[e-2], q[e-1]) \geq slope(q[e-1], i)$，则说明若 $q[e-2]$ 被弹出，$q[e-1]$ 一定被弹出，所以 $q[e-1]$ 无效，可以先弹出。

从 “形” 的角度理解，单调队列维护了一个下凸壳。

10.5.2 应用

主要过程就是研究决策单调性满足的条件，然后选取适当的数据结构维护信息，快速 dp。

实际应用中需注意以下几点：

- 比较斜率时尽量使用乘法避免精度误差，提高效率，要考虑变号时的符号问题，最好保持分母为正。（~~反正也就两处符号，面向样例编程就行了~~）。
- 若 $a[i]$ 单调，使用单调队列，否则使用 10.9.2所述的决策二分栈/队列。
- 若 $b[i]$ 单调，使用单调队列，否则使用平衡树维护凸壳/CDQ 分治（留坑待补）/李超线段树。

  血泪史：「CEOI2017」Building Bridges

  事实上动态凸壳的维护不是很好处理，因为浮点数的精度问题不好解决。我调了 3 个多小时还是 WA（更悲惨的是总共只 WA 一半，但每组捆绑测试都有测试点 WA）。可以考虑维护动态半平面交，毕竟 HPI 还是比较成熟的，由于半平面只有插入，使用第 19.1.14节所述的二进制分组解决。事实证明 HPI+ 二进制分组法数值稳定性比较好（一遍 AC，速度比动态凸包快，代码比动态凸包短）。

以上内容参考了 MashiroSky 的博客[3]。

[3]斜率优化学习笔记 - MashiroSky https://www.cnblogs.com/MashiroSky/p/6009685.html

10.5.3 树上斜率优化

例题：NOI2014 购票

我原先的做法：维护每条重链的完整单调队列，查询时二分出单调队列的一段，然后二分转移点。最后讨论该段前后不在队列上的点取出暴力转移。虽然最终得到满分，但是这种做法严重依赖于数据强度，容易被 Hack，并且细节很多。~~2019.3.17：为什么又是 rank2。。。~~

10.5.3.1 点分治

考虑树退化成链的情况，由于转移长度有限制且不单调，无法使用单调队列维护。使用 CDQ 分治解决：

1. 递归处理 $[l, m]$
2. 计算 $[l, m]$ 到 $[m+1, r]$ 的转移：
   (a) 将 $[m+1, r]$ 按照更新左边界点降序排序，忽略左边界点超过 m 的点
   (b) 将 $[l, m]$ 从右到左加入，维护凸包，加入点 i 后，二分转移所有左边界点恰好为 i 的点
3. 递归处理 $[m+1, r]$

现在考虑树的情况，树上分治一般使用点分治：

记当前分治过程的重心为 g，连通块的根为 u，节点 u 的还未尝试转移的深度最浅的祖先 top_u：

1. 由于这是有根树，u 所在的子连通块与其它连通块并不平等，且 g 需要从 u 处转移，因此首先递归计算 u 的子连通块。
2. 若 $u \neq g$，此时 g 的所有祖先都已经计算完毕，尝试从 u 到 g 的父亲转移 g。
3. DFS 遍历子连通块的节点，序列化点的编号，记录所属连通块的根 bel。
4. 将节点按照 top 排序，从 u 到 u 的祖先逐个加入凸包，尝试更新 dp 值，同时更新 $top = bel$。
5. 递归分治子连通块。

时间复杂度 $O(n \lg^2 n)$。

10.5.3.2 可持久化单调队列

对于可以使用单调队列解决的树上斜率优化问题，在插入一个点时在该点存储被该点弹出的节点编号，回溯时删除自身，同时恢复被弹出的节点。由于不能确定每个点被弹出和恢复的次数，无法保证这个方法的复杂度。

可以发现在加入一个决策点后，队列只要覆盖一个决策点。只需在修改前记录被覆盖位置的决策点和原队列的长度，回溯时恢复该位置与队列长度。插入时使用二分快速计算覆盖位置。不执行队列的弹出操作，同样使用二分计算转移点（这样也可以应对自变量 x 不单调的情况）。

对于本题，由于还有 l 的限制，如果只维护单个凸包，最优决策点有可能被删去（~~我的做法就是暴力处理这种情况~~）。由于凸包是可并的，可以像线段树那样维护深度区间凸包。查询时每个连续区间都二分找到最优决策点，合并时选取这些决策点的最优解，单次查询时间复杂度 $O(\lg^2 n)$。修改时按照上文所述修改 $O(\lg n)$ 个区间，单次修改时间复杂度 $O(\lg^2 n)$。

该内容参考了 xyz32768[4] 和 Sakits[5]的博客。

10.5.4 CDQ 分治维护凸包

例题:NOI2007 货币兑换

记第 i 天最大收益为 c_i,将其兑换为 AB 券的数量为 (a_i, b_i)。那么 j 买入到 i 卖出的转移就相当于计算 $A_i a_j + B_i b_j$ 的最大值。

有两种分析方法:

- 设 $a_j < a_k$,在转移点 i 时,点 j 比点 k 优蕴含着 $(a_j - a_k)A_i + (b_j - b_k)B_i > 0$,化简为 $\frac{b_j - b_k}{a_j - a_k} < -\frac{A_i}{B_i}$。将 (a, b) 视作点,同上文的分析,这里维护 i 之前所有转移点 (a, b) 的上凸包,二分寻找两边斜率包含 $-\frac{A_i}{B_i}$ 的点作为决策点。

- 将 $A_i a_j + B_i b_j$ 视作点 (a_j, b_j) 到直线 $A_i x + B_i y = 0$ 的距离 * 常数,那么最远点可以使用这条直线的平行线夹逼得到。

上述分析是等价的,由于 a_i 不单调,需要维护动态凸包(这里的 a_i 不是预先知道的,不能用李超树)。考虑使用 CDQ 分治,即每次递归处理完左边的答案后,使用左边的凸包转移右边的点。递归前预排序斜率,在递归时分发到左右区间,这样就保证右边点的斜率是有序的,可以 $O(\quad)$ 扫描转移。至于凸包,可以在回溯时归并排序保证水平序。

上述内容参考了 2008 年陈丹琦的集训队作业《从〈Cash〉谈一类分治算法的应用》。

10.6 四边形不等式优化

在解区间动规题时通常会推出以下状态转移方程:

$$dp[i][j] = min(dp[i][k] + dp[k+1][j]) + w[i][j], i \le k < j$$

此时可考虑使用四边形不等式将 $O(n^3)$ 优化到 $O(n^2)$。

接下来定义两个性质:

区间包含的单调性 对于 $a \le b \le c \le d$,有 $f(b, c) \le f(a, d)$。

四边形不等式 对于 $a \le b \le c \le d$,有 $f(a, d) + f(b, c) \ge f(a, c) + f(b, d)$。

定理 10.1 若 w 函数满足上述两个性质,则 dp 函数满足四边形不等式性质。

定理 10.2 若 dp 函数满足四边形不等式,设 $s(i, j)$ 为 $dp[i][j]$ 的转移点 k,有 $s(i, j) \le s(i, j+1) \le s(i+1, j+1)$。

根据该定理可以缩小转移点的搜索范围:

$$dp[i][j] = min(dp[i][k] + dp[k+1][j]) + w[i][j], s[i][j-1] \le k \le s[i+1][j]$$

复杂度分析 对于每一个左端点,最多扫一遍右边所有点,因此时间复杂度降为 $O(n^2)$。

[4][BZOJ3672][Noi2014] 购票(斜率优化 + 点分治)
https://blog.csdn.net/xyz32768/article/details/82709944

[5]bzoj3672: [Noi2014] 购票(树形 DP+ 斜率优化 + 可持久化凸包)
https://www.cnblogs.com/Sakits/p/8215297.html

优化 计算 $w[i][j]$ 时要考虑区间统计方面的优化(如前缀和)。

以上内容参考了 XDU_Skyline 的博客[6]。

10.7 矩阵快速幂优化

10.7.1 常规矩阵快速幂

若 dp 状态转移方程满足如下形式:

$$dp[i] = \sum_{j=1}^{k} c_i dp[i-j]$$

或对于图满足如下形式:

$$dp[d][i][j] = \sum_{(i,k)\in E,(k,j)\in E} dp[d-1][i][k] \cdot dp[d-1][k][j]$$

则可以使用矩阵快速幂优化。

计算 $k*k$ 的转移矩阵 A, dp 初始值为 $1*k$ 的向量 v_0。构造 A, 使其每乘一次 A, 向量表示的区间后移一格, 那么 $A[i][j]$ 表示其在做一次乘法后将第 i 点的值贡献到第 j 点中的权值。

矩阵乘法满足结合律, 因此可以使用矩阵快速幂进行计算。

例题 「NOI2007」生成树计数

使用基尔霍夫定理计算肯定是不现实的(~~不过 +k=3 时的斐波那契数列可以水到 80分~~)。注意到连边是局部的, 且 k 很小。可以使用状态来表示最近 k 个节点的连通性, 然后枚举树边转移。至于连通性的表示, 可以使用最小表示法 DFS 预处理, 即每个节点对应一个编号, 且 i 必须在 $i+1$ 之前出现, 相同编号的节点连通, 并且这些状态保证了之前的链合法。当 $k=5$ 时, 对应的集合划分数目(贝尔数)为 52, 使用矩阵快速幂转移。

参考代码:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
typedef long long Int64;
const int maxk = 60, mod = 65521;
struct Mat {
    Int64 A[maxk][maxk];
    int n, m;
    Mat(int n, int m) : n(n), m(m) {
        memset(A, 0, sizeof(A));
    }
    Int64* operator[](int id) {
        return A[id];
    }
    const Int64* operator[](int id) const {
        return A[id];
    }
```

[6]动态规划专题小结: 四边形不等式优化 https://blog.csdn.net/u014800748/article/details/45750737

```
    Mat operator*(const Mat& rhs) const {
        Mat res(n, rhs.m);
        for(int i = 0; i < n; ++i)
            for(int k = 0; k < m; ++k)
                for(int j = 0; j < rhs.m; ++j)
                    res[i][j] += A[i][k] * rhs[k][j];
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < rhs.m; ++j)
                res[i][j] %= mod;
        return res;
    }
};
int B[7];
int find(int x) {
    return B[x] == x ? x : B[x] = find(B[x]);
}
bool merge(int u, int v) {
    u = find(u), v = find(v);
    if(u != v) {
        B[u] = v;
        return true;
    }
    return false;
}
int k;
struct State {
    int A[7], code;
    State() : code(0) {
        memset(A, 0, sizeof(A));
    }
    int move(int s, bool check) const {
        int rt[8] = {};
        for(int i = 1; i <= k; ++i) {
            int& x = rt[A[i]];
            if(!x)
                x = i;
            B[i] = x;
        }
        B[k + 1] = k + 1;
        for(int i = 1; i <= k; ++i)
            if(s & (1 << (k - i)))
                if(!merge(i, k + 1))
                    return -1;
        if(check) {
            bool flag = false;
            for(int i = 2; i <= k + 1; ++i)
                if(find(1) == find(i)) {
                    flag = true;
                    break;
                }
```

```
            if(!flag)
                return -1;
        }
        int id[8] = {}, cid = 0, dc = 0;
        for(int i = 2; i <= k + 1; ++i) {
            int& x = id[find(i)];
            if(!x)
                x = ++cid;
            dc = dc * 10 + x;
        }
        return dc;
    }
} S[maxk];
int scnt = 0;
void DFS(int u, int c, State s) {
    if(u == k + 1)
        S[scnt++] = s;
    else {
        for(int i = 1; i <= c + 1; ++i) {
            State cur = s;
            cur.A[u] = i;
            cur.code = s.code * 10 + i;
            DFS(u + 1, std::max(i, c), cur);
        }
    }
}
int getId(int code) {
    int l = 0, r = scnt - 1;
    while(l <= r) {
        int m = (l + r) >> 1;
        if(S[m].code == code)
            return m;
        if(S[m].code < code)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}
int main() {
    Int64 n;
    scanf("%d%lld", &k, &n);
    if(k > n)
        k = n;
    DFS(1, 0, State());
    int end = 1 << k;
    Mat move(scnt, scnt);
    for(int i = 0; i < end; ++i)
        for(int j = 0; j < scnt; ++j) {
            int dst = S[j].move(i, true);
```

```
            if(dst != -1)
                ++move[j][getId(dst)];
        }
    Mat res(1, scnt);
    res[0][scnt - 1] = 1;
    for(int i = 2; i <= k; ++i) {
        Mat cur(1, scnt);
        int cend = 1 << (i - 1);
        for(int j = 0; j < scnt; ++j)
            if(res[0][j]) {
                for(int x = 0; x < cend; ++x) {
                    int dst = S[j].move(x, false);
                    if(dst != -1) {
                        dst = getId(dst);
                        cur[0][dst] =
                            (cur[0][dst] + res[0][j]) %
                            mod;
                    }
                }
            }
        res = cur;
    }
    n -= k;
    while(n) {
        if(n & 1)
            res = res * move;
        n >>= 1, move = move * move;
    }
    printf("%lld\n", res[0][0]);
    return 0;
}
```

10.7.2 矩阵链乘秩分解

若大小为 $n*n$ 的矩阵 A 可表示为大小为 $n*k$ 的矩阵 B 与大小为 $k*n$ 的矩阵 C 的乘积, 其中 $k \ll n$。那么可以将 A 的幂拆开, 错位结合, 计算 $k*k$ 的矩阵 $D = CB$, 对 D 快速幂后计算需要的值**(答案向量为 $v_0AD^{c-1}B$,尽量按需计算结果)**。

例题 bzoj3583 杰杰的女性朋友

使用上述方法优化矩阵快速幂的效率。此外还存在一个问题, 矩阵 A 的 k 次幂求的是走 k 步的方案数的转移矩阵, 但是答案要的矩阵为矩阵幂求和。因此我们可以对于每次询问再加一个累加计数器, 自己向自己连边, 对应点向自己连边, 最后单独求出起点到自己的方案数。即再开一个 "信道"?, 然后在新开的信道上, 终点的出边, 计数器的入边与出边均设为 1。

还有另一种方法:注意到这里求的是矩阵的等比数列之和。可以将数列对半分, 然后可以提出 $(1 + A^d)$ 的因子, 子问题的规模减半。细节比较多, 这里不详细写。

参考代码:

```
#include <climits>
#include <cstdio>
```

```
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int maxn = 1005, maxk = 25, mod = 1000000007;
typedef long long Int64;
#define asInt64 static_cast<Int64>
const Int64 end =
    LLONG_MAX - asInt64(mod - 1) * (mod - 1);
struct Mat {
    Int64 A[maxk][maxk];
    int n;
    Mat(int n) : n(n) {
        memset(A, 0, sizeof(A));
    }
    const Int64* operator[](int id) const {
        return A[id];
    }
    Int64* operator[](int id) {
        return A[id];
    }
    Mat operator*(const Mat& rhs) const {
        Mat res(n);
        for(int i = 0; i <= n; ++i)
            for(int k = 0; k <= n; ++k) {
                Int64 mul = A[i][k];
                if(mul == 0)
                    continue;
                for(int j = 0; j <= n; ++j) {
                    res[i][j] += mul * rhs[k][j];
                    if(res[i][j] > end)
                        res[i][j] %= mod;
                }
            }
        for(int i = 0; i <= n; ++i)
            for(int j = 0; j <= n; ++j)
                res[i][j] %= mod;
        return res;
    }
};
Mat powm(Mat A, int k) {
    Mat res(A.n);
```

```
    for(int i = 0; i <= A.n; ++i)
        res[i][i] = 1;
    while(k) {
        if(k & 1)
            res = res * A;
        k >>= 1, A = A * A;
    }
    return res;
}
int out[maxn][maxk], in[maxk][maxn];
Mat buildMat(int N, int K) {
    Mat res(K);
    for(int i = 0; i <= K; ++i)
        for(int k = 0; k <= N; ++k) {
            Int64 mul = in[i][k];
            if(mul == 0)
                continue;
            for(int j = 0; j <= K; ++j) {
                res[i][j] += mul * out[k][j];
                if(res[i][j] > end)
                    res[i][j] %= mod;
            }
        }
    for(int i = 0; i <= K; ++i)
        for(int j = 0; j <= K; ++j)
            res[i][j] %= mod;
    return res;
}
int main() {
    int n = read();
    int k = read();
    for(int i = 1; i <= n; ++i) {
        for(int j = 1; j <= k; ++j)
            out[i][j] = read() % mod;
        for(int j = 1; j <= k; ++j)
            in[j][i] = read() % mod;
    }
    int m = read();
    in[0][0] = out[0][0] = 1;
    for(int t = 0; t < m; ++t) {
        int u = read();
        int v = read();
        int d = read();
        if(d == 0)
            puts(u == v ? "1" : "0");
        else if(d == 1) {
            Int64 res = u == v;
            for(int i = 1; i <= k; ++i)
                res = (res +
                       asInt64(out[u][i]) * in[i][v]) %
```

```
                    mod;
                printf("%lld\n", res);
            } else {
                out[v][0] = 1;
                Mat base = buildMat(n, k);
                Mat pwb = powm(base, d);
                Int64 res = 0;
                for(int i = 0; i <= k; ++i)
                    res =
                        (res +
                         asInt64(out[u][i]) * pwb[i][0]) %
                        mod;
                printf("%lld\n", res);
                out[v][0] = 0;
            }
        }
        return 0;
    }
```

10.7.3 dp 步伐不一致时的解决方案

例题 LOJ#2180.「BJOI2017」魔法咒语

如果单次转移最多需要跳跃 k 步(k 为小常数),可以给每个状态 S_0 引入 $k-1$ 个“延迟状态” S_i,若有状态 S 跳跃 k 步到达状态 T,则实际转移为 $S_0 \to T_{k-1} \to T_{k-2} \to \cdots \to T_0$。注意后面的转移链是固定的, 可以单独预处理。而第一个转移与原来的处理方式相同, 只要根据跳跃步数计算需要延迟转移的时间, 然后连到链上对应的节点。统计时仍然只统计 S_0, 因为延迟状态的贡献是不完整的。

10.7.4 矩阵对角化加速快速幂

有时会推出一些比较简单的矩阵(尤其是三角矩阵与对称矩阵), 这时可以快速找到矩阵的特征值, 计算出特征向量, 将矩阵表示为 $A = PDP^{-1}$ 的形式, 其中 P 是可逆矩阵, 由特征向量组成, D 为对应特征值组成的对角矩阵。

那么 A^n 可以表示为 PD^nP^{-1} 的形式, 中间的部分可以快速幂求出, 两边的部分要根据具体情况讨论。

例题 CF923E Perpetual Subtraction

记初始向量为 P_0, 转移矩阵为 A, 若用矩阵左乘表示转移, 则答案为 A^nP_0。

现在写出转移矩阵 A:

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+1} \\ & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n+1} \\ & & \frac{1}{3} & \cdots & \frac{1}{n+1} \\ & & & \ddots & \vdots \\ & & & & \frac{1}{n+1} \end{bmatrix}$$

这是一个上三角矩阵, 不过即使有稀疏矩阵优化, $O(n^3 \lg M)$ 的快速幂仍然无法承受。

由于该矩阵的特殊性, 考虑对角化该矩阵以加速矩阵幂的计算。易知该矩阵的特征向量为 $1, \frac{1}{2}, \cdots, \frac{1}{n+1}$。接下来代入 $Av = \lambda v$ 求出特征向量 v 并组出 P: 经过小矩阵的推导

得特征值 $\frac{1}{1+x}$ 的特征向量为 $[(-1)^i\binom{i}{0},(-1)^{i+1}\binom{i}{1},\cdots(-1)^{i+n}\binom{i}{n}]^T$。继续打表求出逆矩阵, 然后将矩阵乘法展开, 得到卷积的形式, 使用 NTT 解决。具体证明参见参考链接。

该方法参考了 yhx-12243 的博客[7]。

对角化很简单, 但是寻找特征向量与逆矩阵是困难的。

10.7.5 固定转移矩阵多询问 DP

需求:Luogu P5107 能量采集

记矩阵大小为 n, 最大转移步数为 t, 询问次数为 q。

在多组询问的情况下, 发现每次都会重复计算 2^k 次转移, 可以将这些转移矩阵预处理, 总时间复杂度为 $O(\lg tn^3+q\lg tn^2)$。预处理和询问的复杂度不平衡, 因此还有很大的优化空间。

考虑不使用二进制拆分, 而是使用 k 进制拆分, 那么时间复杂度变为 $O(\log_k t(k-1)n^3+q\log_k tn^2)$, 引入的因子 $k-1$ 可以用于平衡复杂度。在空间允许的情况下, 也可以使用根号分块。

如果询问不强制在线, 还可以将询问按照步长排序, 然后相邻项之间递推。

Warning:对于比较简单的递推式(比如 $f_n=af_{n-1}+bf_{n-2}$),考虑使用特征方程求解,进一步减小常数。

该方法参考了小粉兔的博客[8]。

10.8 动态 dp

动态 dp 与常规 dp 的区别就是加上了多次修改与询问。

序列 DDP 一般使用线段树维护区间信息, 这里不详述。下文的重点是树上 DDP。

10.8.1 常规树上 DDP 方法

树上 DDP 一般使用线段树 + 树链剖分 + 矩阵乘法。

考虑动态最大带权独立集问题:

10.8.1.1 描述 dp 转移

记 f_u 为不选点 u 的子树最大带权独立集, g_u 为选点 v 的子树最大带权独立集。

那么对于点 u 的子树 T_u 有

$$\begin{aligned} f_u &= \sum_{(u,v)\in T_u} max(f_v,g_v) \\ g_u &= w_u+\sum_{(u,v)\in T_u} f_v \end{aligned}$$

类似于算法导论 [4] 中解决所有节点对最短路径问题时介绍的类矩阵乘法, 这里把对单个儿子 v 的转移视为左乘一个由点 u 当前 dp 状态决定的转移矩阵, 把 (f_v,g_v) 视作向量。其中矩阵乘法的定义需要修改:乘法变为加法, 加法变为取 max, 这个新的乘法操作仍然满足结合律。单位阵 I_n 的主对角线上为 0, 其余元素为 $-\infty$。

[7][Codeforces923E/947E]Perpetual Subtraction
https://yhx-12243.github.io/OI-transit/records/cf923E%3Bcf947E.html

[8]题解 P5107【能量采集】- PinkRabbit - 洛谷博客
https://www.luogu.org/blog/PinkRabbit/solution-p5107

所以转移式如下：

$$\begin{bmatrix} f_u & f_u \\ g_u & -\infty \end{bmatrix} \begin{bmatrix} f_v \\ g_v \end{bmatrix} = \begin{bmatrix} f'_u \\ g'_u \end{bmatrix}$$

考虑一条链的情况，链头的 dp 向量即为链上所有点按顺序组成的矩阵链之积（不是向量？视作在链尾后再加一个空点，即向量 $\mathbf{0}$）。每个点的矩阵均为考虑自身权值与分支后的转移矩阵。

此时已经将 dp 转移转化为矩阵乘法的形式。

注意事项

- 「NOIP2018」保卫王国：构造出矩阵后转移矩阵内一定有一列与向量的表示相同，取 dp 值时从这里取。更准确地说，要乘上末尾的哨兵向量观察取值。
- 「SDOI2017」切树游戏：在构造矩阵时不要吝啬矩阵大小，可以加入一些常量辅助转移，然后模拟多次矩阵相乘，找出不变量并省略（尤其是常量所在位置），以节省时间和空间。

10.8.1.2 修改与查询

暴力算法每一次都要维护从修改点到根的点权，转化为矩阵乘法后也不例外。考虑对整棵树进行树链剖分，暴力跳重链维护轻儿子对父亲的贡献，查询时线段树查询区间矩阵乘积，把自己所在重链的后代的贡献施加到自身矩阵，计算出真实的转移矩阵。注意跳重链时根据树链剖分的性质，该操作不超过 $O(\lg n)$ 次，因此每次修改的复杂度为 $O(\lg^2 n)$。

代码如下：

```
#include <cstdio>
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
void printImpl(int x) {
    if(x >= 10)
        printImpl(x / 10);
    putchar('0' + x % 10);
}
void print(int x) {
    if(x < 0)
        putchar('-'), x = -x;
    printImpl(x);
    putchar('\n');
}
int maxi(int a, int b) {
```

```
    return a > b ? a : b;
}
const int size = 100005, inf = 1 << 29;
struct Mat {
    int A[2][2];
    int* operator[](int i) {
        return A[i];
    }
    const int* operator[](int i) const {
        return A[i];
    }
    Mat operator*(const Mat& rhs) const {
        Mat res;
        for(int i = 0; i < 2; ++i)
            for(int j = 0; j < 2; ++j)
                res[i][j] = maxi(A[i][0] + rhs[0][j],
                                 A[i][1] + rhs[1][j]);
        return res;
    }
} T[size << 2];
struct Vec {
    int f, g;
} A[size];
int V[size];
Vec update(Vec a, const Vec& b) {
    a.f += maxi(b.f, b.g);
    a.g += b.f;
    return a;
}
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int p[size], son[size];
int buildTree(int u) {
    int siz = 1, ms = 0;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p[u]) {
            p[v] = u;
            int sv = buildTree(v);
            siz += sv;
            if(sv > ms)
                son[u] = v, ms = sv;
        }
```

```
    }
    return siz;
}
int top[size], id[size], pid[size], icnt = 0, ch[size];
Vec buildChain(int u) {
    int uid = ++icnt;
    id[u] = uid;
    pid[uid] = u;
    A[uid].g = V[u];
    Vec lazy;
    if(son[u]) {
        top[son[u]] = top[u];
        lazy = buildChain(son[u]);
        ch[u] = ch[son[u]] + 1;
    }
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(!top[v]) {
            top[v] = v;
            A[uid] = update(A[uid], buildChain(v));
        }
    }
    Vec res = son[u] ? update(A[uid], lazy) : A[uid];
    return res;
}
#define ls l, m, id << 1
#define rs m + 1, r, id << 1 | 1
void update(int id) {
    T[id] = T[id << 1] * T[id << 1 | 1];
}
int lid[size];
void build(int l, int r, int id) {
    if(l == r) {
        lid[pid[l]] = id;
        Mat& base = T[id];
        base[0][0] = base[0][1] = A[l].f;
        base[1][0] = A[l].g;
        base[1][1] = -inf;
    } else {
        int m = (l + r) >> 1;
        build(ls);
        build(rs);
        update(id);
    }
}
Mat mat;
void updateMat(int u) {
    int id = lid[u];
    T[id] = mat;
    while(id >= 2)
```

```
        update(id >>= 1);
    }
    int nl, nr;
    Mat query(int l, int r, int id) {
130     if(nl <= l && r <= nr)
            return T[id];
        else {
            int m = (l + r) >> 1;
            if(nl <= m && m < nr)
                return query(ls) * query(rs);
            if(nl <= m)
                return query(ls);
            return query(rs);
        }
140 }
    int n;
    Mat getMat(int u) {
        nl = id[u], nr = id[u] + ch[u];
        return query(1, n, 1);
    }
    int main() {
        n = read();
        int m = read();
        for(int i = 1; i <= n; ++i)
150         V[i] = read();
        for(int i = 1; i < n; ++i) {
            int u = read();
            int v = read();
            addEdge(u, v);
            addEdge(v, u);
        }
        buildTree(1);
        top[1] = 1;
        buildChain(1);
160     build(1, n, 1);
        while(m--) {
            int x = read();
            int y = read();
            {
                mat = T[lid[x]];
                mat[1][0] += y - V[x];
                V[x] = y;
            }
            while(true) {
170             int tx = top[x], ptx = p[tx];
                if(ptx) {
                    Mat old = getMat(tx);
                    updateMat(x);
                    Mat now = getMat(tx);
                    {
```

```
                    mat = T[lid[ptx]];
                    int delta =
                        maxi(now[0][0], now[1][0]) -
                        maxi(old[0][0], old[1][0]);
                    mat[0][0] = (mat[0][1] += delta);
                    mat[1][0] += now[0][0] - old[0][0];
                }
                x = ptx;
            } else
                break;
        }
        updateMat(x);
        Mat res = getMat(1);
        print(maxi(res[0][0], res[1][0]));
    }
    return 0;
}
```

为了保证更新轻儿子对父亲矩阵的贡献的复杂度, 不能重新对所有轻儿子 dp, 而应该把自己未修改前的**真实转移矩阵**对父亲转移矩阵的贡献扣去, 然后施加变换更新线段树, 重新计算自身真实转移矩阵然后修改父亲的矩阵。所以对矩阵的修改需要 "延迟更新"。

对于其它动态树形 dp 问题, 其关键是将 dp 转化为矩阵乘法的形式, 然后套树链剖分 + 线段树解决。

上述内容参考了小蒟蒻 yyb 的博客[9]。

10.8.2 全局平衡二叉树

这个黑科技出自 2007 年 Yang Zhe 的论文《SPOJ375 QTREE 解法的一些研究》[11]。

考虑把树链剖分 + 线段树换成 LCT, 直接维护 dp 值可以把查询时间复杂度降到 $O(m \lg n)$。但是 LCT 的常数太大, 其表现不如树链剖分 + 线段树。事实上我们并不需要 LCT 的 link-cut 功能, 因此可以考虑把树静态化, 即构造一个能够暴力向上更新的数据结构。

全局平衡二叉树就能满足这一要求。建树过程很简单:

1. 树链剖分求出重儿子;
2. 给每个节点附上权值, 权值为轻儿子子树大小之和 +1。
3. 对于每条重链找整条链的带权重心, 把重心当做 bst 的根, 然后递归两边继续找带权重心建 bst。

暴力更新即自底向上沿着 bst 和虚边更新。注意我们只需要维护每棵 bst 的先序遍历矩阵积, 对应每条重链的矩阵积。可以发现无论经过的是 bst 的边还是虚边, 子树大小至少增大 1 倍(经过 bst 的情况可类比点分治性质 11.9)。所以树高为 $O(\lg n)$, 查询复杂度降为 $O(m \lg n)$。

模板:

```
#include <cstdio>
int read() {
    int res = 0, c;
    bool flag = false;
```

[9]动态 dp http://www.cnblogs.com/cjyyb/p/10031947.html

```
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
void printImpl(int x) {
    if(x >= 10)
        printImpl(x / 10);
    putchar('0' + x % 10);
}
void print(int x) {
    if(x < 0)
        putchar('-'), x = -x;
    printImpl(x);
    putchar('\n');
}
int maxi(int a, int b) {
    return a > b ? a : b;
}
const int size = 100005, inf = 1 << 29;
struct Mat {
    int A[2][2];
    int* operator[](int i) {
        return A[i];
    }
    const int* operator[](int i) const {
        return A[i];
    }
    Mat operator*(const Mat& rhs) const {
        Mat res;
        for(int i = 0; i < 2; ++i)
            for(int j = 0; j < 2; ++j)
                res[i][j] = maxi(A[i][0] + rhs[0][j],
                                 A[i][1] + rhs[1][j]);
        return res;
    }
    int maxv() const {
        return maxi(A[0][0], A[1][0]);
    }
};
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v) {
```

```
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int p[size], son[size], w[size];
int buildTree(int u) {
    int siz = 1, ms = 0;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p[u]) {
            p[v] = u;
            int sv = buildTree(v);
            siz += sv;
            if(sv > ms)
                son[u] = v, ms = sv;
        }
    }
    w[u] = siz - ms;
    return siz;
}
struct Node {
    int ls, rs;
    Mat mat, mul;
} T[size];
int fa[size];
bool isRoot(int u) {
    int p = fa[u];
    return T[p].ls != u && T[p].rs != u;
}
#define ls T[u].ls
#define rs T[u].rs
void update(int u) {
    if(ls && rs)
        T[u].mul = T[ls].mul * T[u].mat * T[rs].mul;
    else if(ls)
        T[u].mul = T[ls].mul * T[u].mat;
    else if(rs)
        T[u].mul = T[u].mat * T[rs].mul;
    else
        T[u].mul = T[u].mat;
}
int ch[size], sw[size];
int buildChainImpl(int l, int r) {
    if(l > r)
        return 0;
    int sum = 0, tot = sw[r] - sw[l - 1];
    for(int i = l; i <= r; ++i) {
        int u = ch[i];
        sum += w[u];
        if(sum * 2 >= tot) {
```

```
            ls = buildChainImpl(l, i - 1);
            rs = buildChainImpl(i + 1, r);
            fa[ls] = fa[rs] = u;
            update(u);
            return u;
        }
    }
}
bool flag[size];
int buildChain(int u) {
    for(int i = u; i; i = son[i])
        flag[i] = true;
    for(int i = u; i; i = son[i]) {
        for(int j = last[i]; j; j = E[j].nxt) {
            int v = E[j].to;
            if(!flag[v]) {
                int p = buildChain(v);
                fa[p] = i;
                T[i].mat[0][0] = T[i].mat[0][1] +=
                    T[p].mul.maxv();
                T[i].mat[1][0] += T[p].mul[0][0];
            }
        }
    }
    int csiz = 0;
    for(int i = u; i; i = son[i]) {
        ch[++csiz] = i;
        sw[csiz] = sw[csiz - 1] + w[i];
    }
    return buildChainImpl(1, csiz);
}
int V[size];
int main() {
    int n = read();
    int m = read();
    for(int i = 1; i <= n; ++i)
        V[i] = read();
    for(int i = 1; i < n; ++i) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    buildTree(1);
    for(int i = 1; i <= n; ++i) {
        T[i].mat[1][0] = V[i];
        T[i].mat[1][1] = -inf;
    }
    int root = buildChain(1);
    while(m--) {
```

```
        int x = read();
        int y = read();
        T[x].mat[1][0] += y - V[x];
        V[x] = y;
        for(int i = x; i; i = fa[i])
            if(fa[i] && isRoot(i)) {
                int p = fa[i];
                T[p].mat[0][0] = T[p].mat[0][1] -=
                    T[i].mul.maxv();
                T[p].mat[1][0] -= T[i].mul[0][0];
                update(i);
                T[p].mat[0][0] = T[p].mat[0][1] +=
                    T[i].mul.maxv();
                T[p].mat[1][0] += T[i].mul[0][0];
            } else
                update(i);
        print(T[root].mul.maxv());
    }
    return 0;
}
```

这种写法算法速度快、代码简单、易于理解, 推荐使用此法。
上述内容参考了 shadowice1984 的博客[10]。

10.8.3 子树 DP 值查询

如果需要向 bzoj4712 洪水那样查询某个子树内的 dp 值, 不能直接使用子树根节点的信息。因为整条链是一棵二叉树, 这个根节点的矩阵积还包括其左儿子的贡献, 但这一部分并不是它的子树。实际答案应该为自身及其后继的积, 实现时只有往右移才算入这个祖先的贡献。

参考代码:

```
Mat res;
bool init = false;
int last = T[x].l;
while(true) {
    if(T[x].l == last) {
        if(init)
            res = res * T[x].mat;
        else
            res = T[x].mat, init = true;
        if(T[x].r)
            res = res * T[T[x].r].mul;
    }
    last = x;
    if(isRoot(x))
        break;
    else
```

[10]题解 P4643【【模板】动态 dp】- 某菜鸡的 blog
https://www.luogu.org/blog/ShadowassIIXVIIIIV/solution-p4643

```
        x = T[x].p;
}
```

10.8.4 固定思路

1. 转移方式：推导 dp 转移方程，确定最少需要的信息，组成向量
2. 矩阵表达：将父亲 u 转移儿子 v 的信息写成矩阵左乘向量的形式，矩阵与 u 相关，向量与 v 相关。若转移无法被表达，则尝试给向量增加新的维度（而不是让矩阵变为非方阵），新维度的信息用常量填充（$0, 1, \pm\infty$）。
3. 确定有效项：模拟多次矩阵乘法，直至不变量的个数稳定（即这个由变量组成的多元组满足乘法封闭）。省去不变量，硬编码矩阵乘法。对于相等项可以只保留一份（**必须保证正确性**）。
4. 结果读取：构造链尾哨兵，右乘转移矩阵（使用已确定有效项的矩阵而不是最初的转移矩阵），根据向量元素的实际意义得到结果。
5. 轻儿子 DP 值转移到重链上的方式：最好满足可减性（最好不要是乘法，可能出现没有逆元的情况）。注意 + with max/min 意义下乘法时不要为了使用 int 而给 + 操作做 clamp，因为在替换时需要做信息减法，结果是错误的。对于不满足信息可减性的，与 LCT 维护子树信息的做法一样，给每个点开一个数据结构维护轻儿子的信息和。

   Update：由于每个点最多只给一个父亲贡献自己作为轻儿子的信息，可以将每个节点的将自己与所有轻儿子平铺到序列上，记录其区间，然后使用线段树维护。
6. 单位阵构造（Optional，用于子树 DP 值查询）：按照方程模拟确定 I 的每一项，如果不存在单位阵，乘法时记录是否已初始化的 flag。

10.9 决策单调性 DP

斜率优化与四边形不等式是决策单调性 DP 的特殊情形，参见第 10.5节与第 10.6节。

在分析、实现、Debug 难度、错误率方面，分治 DP< 决策二分栈 DP< 斜率优化 DP。在题目条件与运行时间允许的情况下，优先使用分治 DP。

10.9.1 双指针优化

对于 dp 方程为 $f_i = max/min\{w_{j,i}\}$ 的问题，记 f_i 的转移决策点为 P_i，若对于任意 $i < j$，满足 $P_i < P_j$，且对于任意 i 的可行转移点集合 P，有 $P_i = P_{min}$，那么可以维护 2 个指针，一个指示 i，一个指示转移决策点，当不在可行转移点集合时才移动决策点，时间复杂度 $O(n)$。

10.9.2 决策二分栈 DP

该方法适用于 dp 方程为 $f_i = max/min\{g_j + w_{j,i}\}$，其中 g_j 是一些与 j 有关的函数，可能包含 f_j，且对于任意 $i < j$，满足 $P_i < P_j, w_{j,i}$ 可以快速算出。

由于这类题无法贪心，必须存储决策转移点。不过注意到 i 对应的决策转移点随着 i 的递增而递增，反过来每个决策转移点映射了一段 i 的区间。那么可以维护一个单调栈/队列，内部元素均为当前有效决策转移点（还未计算转移的位置落在它们的管辖区间内）。转移点之间维护一个分界点，使用二分计算（若函数较简单也可以直接求交点，还可以用牛顿迭代法）。添加新转移点时计算新点与栈顶/队尾的转移点的分界点，若产生交叉则将其弹出。计算转移决策点时也使用二分计算。时间复杂度 $O(n \lg n)$。

在二分决策点 $x < y$ 的分界点时, 左端点可设为 y 而不是 1, 因为 f_i 只可能从 $j \le i$ 的位置转移。**注意对于连续决策函数，即使其 DP 值需要取整，二分分界点时也要使用浮点计算。**

例题 [NOI2009] 诗人小 G

参考代码:

```
#include <cmath>
#include <cstdio>
#include <cstring>
const int size = 100005, maxk = 31;
int len[size], L, P;
typedef long double FT;
const FT maxl = 1e18;
FT dp[size];
FT powf(FT delta, int k) {
    FT res = 1;
    while(k) {
        if(k & 1)
            res = res * delta;
        k >>= 1, delta = delta * delta;
    }
    return res;
}
FT calc(int i, int j) {
    FT delta = len[j] - len[i] - L;
    return dp[i] + powf(fabsl(delta), P);
}
int k[size], q[size], n;
int getEdge(int x, int y) {
    int l = x, r = n + 1, res = 0;
    while(l <= r) {
        int m = (l + r) >> 1;
        if(calc(x, m) <= calc(y, m))
            res = m, l = m + 1;
        else
            r = m - 1;
    }
    return res + 1;
}
char buf[size][maxk];
int trans[size], pos[size];
int main() {
    int t;
    scanf("%d", &t);
    for(int x = 1; x <= t; ++x) {
        scanf("%d%d%d", &n, &L, &P);
        ++L;
        for(int i = 1; i <= n; ++i) {
            scanf("%s", buf[i]);
```

```
            len[i] = len[i - 1] + strlen(buf[i]) + 1;
        }
        int b = 1, e = 1;
        q[1] = 0;
        for(int i = 1; i <= n; ++i) {
            while(b < e && k[b] <= i)
                ++b;
            int p = q[b];
            trans[i] = p;
            dp[i] = calc(p, i);
            while(b < e &&
                  getEdge(q[e], i) <= k[e - 1])
                --e;
            k[e] = getEdge(q[e], i);
            q[++e] = i;
        }
        if(dp[n] > maxl)
            puts("Too hard to arrange");
        else {
            printf("%.0Lf\n", dp[n]);
            int cnt = 0;
            for(int i = n; i; i = trans[i])
                pos[++cnt] = i;
            pos[cnt + 1] = 0;
            for(int i = cnt; i >= 1; --i) {
                for(int j = pos[i + 1] + 1; j < pos[i];
                    ++j)
                    printf("%s ", buf[j]);
                puts(buf[pos[i]]);
            }
        }
        printf("--------------------");
        if(x != t)
            putchar('\n');
    }
    return 0;
}
```

上述内容参考了 FlashHu 的博客[11]。

10.9.3 分治 DP

该方法适用于 f_i 之间独立且决策二分栈 DP 中 $w_{j,i}$ 无法快速计算或不好写的情况。

例题:「雅礼集训 2017 Day5」珠宝

这是一个经典的背包问题, 但是按照背包做会 TLE。发现代价 c 很小, 考虑按照 c 分类然后转移, 花费 kc 的代价时贪心地选取价值前 k 大的物品(我的思路止步于此)。

由于同一条 dp 依赖链上的位置模 c 同余, 可以考虑再按照位置模 c 分类, 将每一条链拆开处理。注意到选取前 k 大物品的代价的增长率是单调非增的, 尝试验证该 dp 是否

[11]DP 的各种优化(动态规划, 决策单调性, 斜率优化, 带权二分, 单调栈, 单调队列)
https://www.cnblogs.com/flashhu/p/9480669.html

有决策单调性。

采用反证法, 设两个同类 dp 点 i, j, 满足 $i < j$, 记它们的转移决策点分别为 P_i, P_j, 满足 $P_i > P_j$。记原先的 dp 数组为 dp, 前 k 大前缀和数组为 w。根据决策点的定义, 有:

$$\begin{aligned} dp[P_i] + w[i - P_i] &> dp[P_j] + W[i - P_j] \\ \Rightarrow dp[P_i] - dp[P_j] &> W[i - P_j] - w[i - P_i] \\ dp[P_i] + w[j - P_i] &\leq dp[P_j] + W[j - P_j] \\ \Rightarrow dp[P_i] - dp[P_j] &\leq W[j - P_j] - w[j - P_i] \end{aligned}$$

由不等式传递性可得 $W[i - P_j] - w[i - P_i] < W[j - P_j] - w[j - P_i]$, 由于 $i < j$ 且增长率单调非增, 与该式产生矛盾, 因此转移点是单调的。

那么可以写一个分治程序 $solve(l, r, L, R)$, 表示处理 $[l, r]$ 之间的 dp 值, 转移区间在 L, R。每次在转移区间内扫一遍求出 mid 的 dp 值, 得到转移点, 最后左右递归处理, 时间复杂度 $O(ck \lg k)$。

代码:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
std::vector<int> A[305];
bool cmp(int a, int b) {
    return a > b;
}
const int maxk = 50005;
typedef long long Int64;
Int64 dp[maxk], w[maxk], old[maxk];
int lut[maxk], num;
void solve(int l, int r, int pl, int pr) {
    if(l > r)
        return;
    int m = (l + r) >> 1, trans = 0,
        end = std::min(pr, m);
    Int64 maxv = 0;
    for(int i = std::max(pl, m - num); i <= end; ++i) {
        Int64 val = old[lut[i]] + w[m - i];
        if(val > maxv)
            trans = i, maxv = val;
```

```
    }
    dp[lut[m]] = maxv;
    solve(l, m - 1, pl, trans);
    solve(m + 1, r, trans, pr);
}
int main() {
    int n = read();
    int k = read();
    for(int i = 1; i <= n; ++i) {
        int c = read();
        A[c].push_back(read());
    }
    for(int i = 1; i <= 300; ++i)
        if(A[i].size()) {
            memcpy(old + 1, dp + 1, sizeof(Int64) * k);
            int end = std::min(
                static_cast<int>(A[i].size()), k / i);
            std::partial_sort(A[i].begin(),
                              A[i].begin() + end,
                              A[i].end(), cmp);
            for(int j = 0; j < end; ++j)
                w[j + 1] = w[j] + A[i][j];
            num = end;
            for(int j = 0; j < i; ++j) {
                int cnt = 0, now = j;
                while(now <= k) {
                    lut[++cnt] = now;
                    now += i;
                }
                solve(2, cnt, 1, cnt);
            }
        }
    for(int i = 1; i <= k; ++i)
        printf("%lld ", dp[i]);
    return 0;
}
```

上述内容参考了 ShichengXiao 的博客[12]。

注意 $w_{j,i}$ 无法快速算出但是可以快速转移(类似莫队)时,要预处理一部分区间以保证分治复杂度。**准确地说,分治复杂度要严格与决策区间长度 + 处理区间长度相关。**

例题:CF868F Yet Another Minimization Problem

推出 DP 方程,滚动第一维,发现转移具有决策单调性,并且无法快速计算 $w_{j,i}$,考虑使用分治解决。记 $solve(l,r,b,e)$ 表示计算 $[l,r]$ 的 dp 值,决策区间在 $[b,e]$,记 $m=(l+r)/2$。原做法:记 $end=min(e,m)$,暴力加入 $(end,m]$ 内的元素,然后从 end 到 n 边计算 dp 边添加新元素。复杂度不保的地方在于每次都暴力加入 $(end,m]$ 内的元素。

正确姿势:注意到 $b<l$,向左递归时都要用到区间 $[b,l)$,考虑每次调用 $solve$ 前都已加入 $[b,l)$ 的元素。接下来分析 3 个子过程,记单层分治复杂度中与处理区间长度相关的项为 A,与决策区间长度相关的项为 B:

[12]DP 及其优化
https://www.cnblogs.com/ShichengXiao/p/9501386.html

- 寻找 m 的决策点 p: 该过程需要用到区间 $[b, m]$, 首先再加入 $[l, m]$ 的元素, 然后从 b 开始移动向 end 缩减区间计算 dp, 最后还原区间 $[b, end]$, 删除 $[l, m]$。时间复杂度 $A + B + B + A$。
- 准备左递归要预处理的区间: 预处理部分已就绪, 直接递归。
- 准备右递归要预处理的区间: 需要将区间 $[b, l)$ 迁移至区间 $[p, m)$, 这个过程可视为左端点和右端点的移动, 因此复杂度仍然是靠谱的。最后仍然要退回 $[b, l)$ 给调用端用, 时间复杂度 $B + A + A + B$。

参考代码:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005;
typedef long long Int64;
const Int64 inf = 1LL << 60;
int cnt[size], A[size];
Int64 cur, dp[2][size], *f, *g;
void solve(int l, int r, int b, int e) {
    if(l > r)
        return;
    int m = (l + r) >> 1, end = std::min(e, m), tp;
    for(int i = l; i <= m; ++i)
        cur += cnt[A[i]]++;
    Int64 minv = inf;
    for(int i = b; i <= end; ++i) {
        Int64 val = cur + f[i - 1];
        if(val < minv)
            minv = val, tp = i;
        cur -= --cnt[A[i]];
    }
    g[m] = minv;
    for(int i = b; i <= end; ++i)
        cur += cnt[A[i]]++;
    for(int i = l; i <= m; ++i)
        cur -= --cnt[A[i]];
    solve(l, m - 1, b, tp);
    int sa = m - l + 1 + tp - b,
```

```
        sb = l - b + m - tp + 1;
    if(sa < sb) {
        for(int i = l; i <= m; ++i)
            cur += cnt[A[i]]++;
        for(int i = b; i < tp; ++i)
            cur -= --cnt[A[i]];
    } else {
        for(int i = b; i < l; ++i)
            --cnt[A[i]];
        cur = 0;
        for(int i = tp; i <= m; ++i)
            cur += cnt[A[i]]++;
    }
    solve(m + 1, r, tp, e);
    if(sa < sb) {
        for(int i = b; i < tp; ++i)
            cur += cnt[A[i]]++;
        for(int i = l; i <= m; ++i)
            cur -= --cnt[A[i]];
    } else {
        for(int i = tp; i <= m; ++i)
            --cnt[A[i]];
        cur = 0;
        for(int i = b; i < l; ++i)
            cur += cnt[A[i]]++;
    }
}
int main() {
    int n = read();
    int k = read();
    f = dp[0], g = dp[1], cur = 0;
    for(int i = 1; i <= n; ++i) {
        A[i] = read();
        cur += cnt[A[i]]++;
        f[i] = cur;
    }
    cur = 0;
    memset(cnt + 1, 0, sizeof(int) * n);
    for(int j = 2; j < k; ++j) {
        solve(1, n, 1, n);
        std::swap(f, g);
    }
    Int64 res = inf;
    for(int i = n; i >= 1; --i) {
        cur += cnt[A[i]]++;
        res = std::min(res, f[i - 1] + cur);
    }
    printf("%lld\n", res);
    return 0;
}
```

该内容参考了 FlashHu 的题解[13]。
小心地控制复写顺序连滚动数组都不用(右-中-左)。

10.9.4 时间复杂度陷阱

如遇到多轮相同的分治 DP, 则考虑将时间复杂度优化到与轮数无关。否则容易被卡常/TLE。

例题:[Apio2014] 序列分割(2019.4.8:调了一下午终于在 bzoj 上变成 rank1)

得出得分与序列切分顺序无关后就可以开始分治 DP 了。进一步考虑答案的实际表达式, 记 b_i 为第 i 块的总和, 有 $ans = \frac{1}{2}\left((\sum b_i)^2 - min\sum b_i^2\right)$。原问题被转化为最小化块的平方和, 但是仍然无法降低时间复杂度, 若去掉因子 k 则无法保证其块数恰好为 k。

不过即使数组存在负数我们也可以求出块平方和的最小值, 以及对应的最小的块数, 使用斜率优化解决, 然后再套 WQS 二分。

综上所述, 当遇到多轮分治 DP 时考虑转化为 WQS 二分。

10.9.5 决策单调性快速判断

该内容参考了 FlashHu 的博客[14]。
定义决策函数 $F_j(i)$ 为从 j 转移到 i 的 dp 值。
具有决策单调性的 DP 有以下特征:

- 决策函数是直线(使用斜率优化)。
- 发现某些部分分决策函数是直线, 可以使用斜率优化做。部分分有一定的提示作用。
- 两个决策函数只有一个交点。
- 决策函数的导函数单调:
  - 导函数单调递增, 求最大值/单调递减, 求最小值:单调栈
  - 导函数单调递增, 求最小值/单调递减, 求最大值:单调队列
- 决策函数可表示为多个满足决策单调性的子决策函数之和。
- 输出决策点发现决策具有单调性。
- 带绝对值的决策函数拆成两种情况后均具有单调性。遇到关于下标之差绝对值的决策函数, 拆成两次 DP 来做, 每次强制单方向转移。

10.10 WQS 二分凸优化

需求:给定 n 个物品, 需要从中选取 C 个物品, 最大/小化权值和。设 $g(x)$ 为选取 x 个物品的最优解, 题目还满足 $g(x)$ 是一个凸函数, 可以快速计算出 $g(x)$ 的最值和取到最值的 x。

因为 $g(x)$ 是凸函数, 所以 $g'(x)$ 单调, 取到最值的 x 就是 $g'(x)$ 的根。如果让根移动到 C, 就可以取得 $g(C)$ 的值。那么可以根据当前的 x 与 C 的大小, 二分 $g'(x)$ 的上下偏移, 使得 $f'(x) = g'(x) + k$ 的根移动到 C。此时对应的代价函数 $f(x) = g(x) + kx$, 最后的答案 $g(C) = f(C) - kC$。

[13]洛谷 CF868F Yet Another Minimization Problem(动态规划, 决策单调性, 分治)
https://www.cnblogs.com/flashhu/p/9495839.html
[14]不失一般性和快捷性地判定决策单调(洛谷 P1912 [NOI2009] 诗人小 G)(动态规划, 决策单调性, 单调队列)
https://www.cnblogs.com/flashhu/p/9521094.html

二分需要求出 $f(x)$ 的极值点, k 的实际意义是每多选一个物品需要额外增加 k 的权值, 可以根据实际意义理解二分时的区间缩小。引入 k 后忽略了个数限制, 可以做更低维的 DP/贪心。这种优化方法不仅在 DP 中使用, 比如 [国家集训队 2]Tree I。

注意这里只需要整数二分。由于 DP 的特殊性, $g(x)$ 的值域为整数, 其图像由许多横向长度为 1 的线段组成。那么 $g'(x)$ 的函数图像就是许多取值为整数的水平线, 因此只需二分整数偏移。

BZOJ5311 贞鱼：注意可能存在连续极值点的情况，此时要求极左或极右的极值点，然后根据极值点是极左还是极右判断二分中更新答案的位置。在本题中求的是极左极值点，因此在极左极值点 $\leq$ 目标点时更新答案。如果在极左极值点 $\geq$ 目标点时更新答案，则无法取到极左极值点到极右极值点覆盖目标点的情况。

CF739E Gosha is hunting：当有两个个数约束时使用二分套二分解决。虽然使用浮点二分，但仍然要注意极值点取极左/极右，控制不等号来控制区间缩小方向。

优化：如果发现极值点恰好是目标点, 直接 break。

上述内容参考了 FlashHu 的博客[15]。

至于二分的范围, 考虑两类极端情况, 即选取所有与选取一个的情况, 调整范围就是它们的极值, 可以根据输入数据计算而不是预置。

除了选取恰好 k 个物品外, 恰好将序列分成 k 段也是典型应用。

10.11 特殊 DP

10.11.1 LCIS

LCIS 即 Longest Common Increasing Subsequence, 最长公共上升子序列。注意这个问题不能用 LCS+LIS 解决。

输入长度为 n 的数组 A 和长度为 m 的数组 B。

10.11.1.1 常规 dp

记 $dp[i][j]$ 为数组 A 的前 i 个元素与数组 B 的前 j 个元素中, 以 $B[j]$ 为结尾的最长公共上升子序列长度。

接下来分类讨论状态转移方程:

- $A[i] = B[j]$, 则考虑将 $A[i]$ 与 $B[j]$ 匹配。显然转移位置的第一维为 $i-1$, 第二维只能枚举 $B[k] < B[j]$ 转移。
- 若不匹配则直接继承 $dp[i-1][j]$ 的值。

综上所述, 状态转移方程为:

$$dp[i][j] = \begin{cases} dp[i-1][j] & \text{if } A[i] \neq B[j] \\ max\{dp[i-1][k]\} + 1 (0 \leq k < j \wedge B[k] < B[j]) & \text{if } A[i] = B[j] \end{cases}$$

时间复杂度 $O(nm^2)$。

10.11.1.2 优化

注意到第一层循环后 $A[i]$ 是固定的, 而枚举取 max 的情况仅在 $A[i] = B[j]$ 时出现。因此可以在第一层循环内维护一个最优转移值, 当 $A[i] > B[j]$ 时更新该值, 当 $A[i] = B[j]$ 时用该值更新, 时间复杂度 $O(nm)$。如果不需要输出方案还可以滚动数组节省空间。

上述内容参考了 ojnQ 的博客[16]。

[15] DP 的各种优化(动态规划, 决策单调性, 斜率优化, 带权二分, 单调栈, 单调队列) https://www.cnblogs.com/flashhu/p/9480669.html

[16] LCIS 最长公共上升子序列问题 DP 算法及优化 https://www.cnblogs.com/WArobot/p/7479431.html

10.11.2 Dilworth 定理

对于一个偏序集, 最少链划分等于最长反链长度。

通俗地讲, 举个例子, 把一个序列划分成最少的上升子序列数等于最长不升子序列长度。

10.11.3 杨氏图表(排序矩阵)

杨氏图表是一个二维表, 若某个位置没有元素, 则它的右边与下边没有元素; 否则若它的右边与下边有元素, 其值比它自身大。

$1 \cdots n$ 组成的杨氏图表个数为数列 A000085, 递推式为 $f(0) = f(1) = 1, f(n) = f(n-1) + (n-1)f(n-2)$。

10.11.3.1 钩长公式

该公式用来计算给定杨氏图表的形状的方案数。

定义某个位置的钩长 h_k 为它下边和它右边的格子数 $+1$, 则方案数为 $\frac{n!}{\prod_{k=1} h_k}$。

10.11.3.2 作为数据结构

查找元素 x 的位置 从矩阵右上角开始寻找, 保证元素 x 要么不存在, 要么在当前元素的左下角。

算法步骤如下:

1. 若当前元素 $= x$, 返回;
2. 若当前元素 $> x$, 向左移动一格, 因为下方的数都比自己大;
3. 若当前元素 $< x$, 向下移动一格, 因为左边的数都比自己小。

记矩阵规模为 $m * n$, 时间复杂度 $O(m+n)$。此时杨氏图表像一个平衡树, 每次选取一个 key 缩小搜索范围。

查找第 k 大值 首先二分值找到大于矩阵内 k 个格子的值 x, 然后在矩阵中找最大的小于 x 的值。计算小于指定数的格子数与找最大的小于指定数的格子均类似上述做法。时间复杂度 $O((m+n)\lg(m+n))$。

上述内容参考了 acdreamers 的博客[17]与 Wikipedia-EN[18]。

10.11.4 GarsiaWachs 算法

该算法用来解决相邻石子合并问题, 与该问题同构的还有最优二叉搜索树问题。该问题有一个 $O(n^2)$ 的朴素区间 dp 解法。

算法步骤如下:

1. 从左往右找到第一个 k, 满足 $w[k-1] \le w[k+1]$, 为了简化算法插入两个哨兵 $w[0] = w[n+1] = \infty$。

[17]杨氏矩阵与钩子公式
https://blog.csdn.net/acdreamers/article/details/14549077
[18]Young tableau
https://en.wikipedia.org/wiki/Young_tableau

2. 合并 $w[k-1]$ 与 $w[k]$, 记新节点的权值为 w_{new}, 向前寻找第一个大于 w_{new} 的位置 j, 将其插入 j 的后面。

3. 迭代直至只剩一个节点。

这个算法的复杂度仍然是 $O(n^2)$ 的。注意每次找到 k 后, 处理它只需要用到之前的信息。那么可以考虑逐个插入元素, 当前的 $k-1$ 满足条件时迭代地维护序列。全部插入完毕后, 由于尾部的 ∞, 尾部的元素会不断被合并。可以发现在任意时刻, 序列都是由一堆单调不增的子序列构成, 使用平衡树维护可以达到 $O(n \lg n)$ 的复杂度(为什么没人写? 难道是 $O(n^2)$ 能过的原因吗?)。

~~这里没有参考代码, 我也不想写平衡树~~。

上述内容参考了 acdreamers 的博客[19]。

10.11.5 双单调性优化

例题: [POI2010]KLO-Blocks

将该问题转化为找到最长的平均数 $\geq k$ 的连续段, 将每个数 $-k$ 再做前缀和 s 后, 问题进一步转化为对于每个 r, 找到最小的 l 满足 $s[r] \geq s[l]$。

注意对于两个决策点 $l < r$, 若 $s[r] \geq s[l]$, 则 r 一定不会被选择。因此可以维护一个单调栈(没有距离限制, 无法移动头部), s 的值随着栈内元素下标的增加而下降。dp 时在单调栈中二分查找。时间复杂度 $O(n \lg n)$。

注意对于两个查询点 $l < r$, 若 $s[r] \geq s[l]$, l 的决策点必定可转移 r, 并且得到更优的答案, 那么 l 就不用查询了。这时需要查询的点也组成一个单调栈, s 的值随着栈内元素下标的增加而下降(注意这两个栈不同, 差别在于对 $s[l] == s[r]$ 的处理, 决策栈保留前者, 而查询栈保留后者)。

发现查询栈中的 s 是下降的, 决策点单调右移, 用双指针扫描支持 $O(n)$ 查询。

该方法参考了 kczno1 的题解[20]。

10.11.6 折半状压 DP

例题:【CTSC2017】吉夫特

使用 lucas 定理可知选取序列的 $a_{b_{k+1}}$ 是 a_{b_k} 的子集。

朴素的做法是记 $dp[i][s]$ 为前 i 位数, 序列尾的子集为 s 的方案数, 更新时将 $dp[i-1][A[i]]+1$ 累加到答案和 $A[i]$ 的子集上。枚举子集可以滚掉第一维, 并且最终答案要扣除 n 除去只有一个数的情况。时间复杂度 $O(na_{max})$, 实际上由于 a 两两不同, 这个复杂度并不满, 通过计算枚举集合数实际上只有 0.003 左右。

DP 还有另一种表达方式: $dp[i][s]$ 为前 i 位数, 序列尾为 s 的方案数, 更新时枚举 $A[i]$ 的父集来获取累加值。

总结两种 DP 方式: 一种方法可以 $O(1)$ 获取累加值, 但需要 $O(2^k)$ 枚举集合更新; 另一种方法需要 $O(2^k)$ 枚举集合获取累加值, 但可以 $O(1)$ 更新。这两个极端启发我们可以使用一种折中的策略。

考虑将状态分成高低位, 记 $dp[i][j][k]$ 为前 i 位数, 序列尾的高位的子集为 j, 低位为 k 的方案数。那么更新时可以 $O(\sqrt{a_{max}})$ 枚举低位的父集获取累加值, 并且以相同的复杂度枚举高位的子集更新。

该内容受到了 cjyyb 的博客的启发(问题不同, 但是策略相同)[21]。

---

[19] 石子合并 (GarsiaWachs 算法)
https://blog.csdn.net/acdreamers/article/details/18043897

[20] 题解 P3503【[POI2010]KLO-Blocks】
https://www.luogu.org/blog/user9168/solution-p3503

[21] 求集合中选一个数与当前值进行位运算的 max
https://www.cnblogs.com/cjyyb/p/9388651.html

10.12 杂记

- 我原先写区间 dp 时首先枚举长度, 然后枚举左端点:

```
0     for(int len=1;len<=n;++len)
          for(int l=1,r=len;r<=n;++r)
```

这样子长度较小的区间先于长度较大的区间 dp 完毕。

不过还有另一种写法:

```
0     for(int l=n;l>=1;--l)
          for(int r=l;r<=n;++r)
```

这种区间枚举方式代码简短, 在缓存友好方面倒是没有什么优势。

Chapter 11

树

11.1 最小公共祖先 339
11.1.1 倍增法 339
11.1.2 树链剖分 340
11.1.3 欧拉序 +ST 表 340
11.1.4 Tarjan 342
11.2 链剖分 343
11.2.1 轻重链剖分 343
11.2.2 长链剖分 345
11.3 树的直径 348
11.3.1 定义与计算 348
11.3.2 性质 348
11.3.3 多子树直径查询 348
11.4 Dsu On Tree 349
11.5 Purfer Sequence 350
11.5.1 构造 Purfer Sequence 350
11.5.2 恢复原树 350
11.5.3 计数应用 350
11.6 虚树 350
11.7 点分治 352
11.7.1 静态点分治 352
11.7.2 动态点分治 366

11.1 最小公共祖先

11.1.1 倍增法

预处理:DFS 时计算每个节点的深度和 2^k 级祖先。

查询:首先将较深节点跳到同一高度,若原节点在一条链上,则较浅的点为 LCA,算法结束。否则按 k 从大到小保持在不跳到同一祖先的前提下尽量向上同时跳。最后这两个节点的父亲就是原节点的 LCA。

```
int d[size],p[size][20];
void DFS(int u) {
    for(int i=1;i<20;++i)
        p[u][i]=p[u][i-1][i-1];
    for(int i=last[u];i;i=E[i].next) {
        int v=E[i].to;
        if(p[u][0]!=v) {
            p[v][0]=u;
            d[v]=d[u]+1;
            DFS(v);
        }
    }
}
int lca(int u,int v){
    if(d[u]<d[v])std::swap(u,v);
    int delta=d[u]-d[v];
    for(int i=0;i<20;++i)
        if(delta&(1<<i))
            u=p[u][i];
    if(u==v)
        return u;
    for(int i=19;i>=0;--i)
        if(p[u][i]!=p[v][i])
            u=p[u][i],v=p[v][i];
    return p[u][0];
}
```

预处理 $O(n\lg n)$, 单次查询 $O(\lg n)$。

11.1.2 树链剖分

树链剖分内容参见第 11.2.1节。树链剖分后, 如果在同一条链上则返回较浅者, 否则令链头较深的节点向上跳。

```
int lca(int u,int v) {
    while(top[u]!=top[v]) {
        if(d[top[u]]>d[top[v]])u=p[top[u]];
        else v=p[top[v]];
    }
    return d[u]<d[v]?u:v;
}
```

两趟 DFS 预处理 $O(n)$。由于树链剖分后重链不超过 $\lg n$ 条, 所以单次查询也是 $O(\lg n)$ 的, 但树链剖分的常数比倍增法小。

11.1.3 欧拉序 +ST 表

考虑 DFS 序, 显然两个来自节点 i 的不同子树的点的 LCA 为节点 i, 那么可以在 DFS 完一棵子树后加入节点 i 的深度作为隔板, 维护 ST 表查询区间内深度最小的节点编号。为了应对点对在一条到根的链上的情况, 同时也为了给节点 i 一个访问时间戳, 处理两个节点为祖孙关系的情况, 在遍历节点 i 之初就插入一个隔板。

```
int icnt = 0, A[size * 2][18], L[size], d[size];
void DFS(int u) {
    A[++icnt][0] = u;
    L[u] = icnt;
    for (int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if (!L[v]) {
            d[v] = d[u] + 1;
            DFS(v);
            A[++icnt][0] = u;
        }
    }
}
int choose(int u, int v) {
    return d[u] < d[v] ? u : v;
}
void buildST() {
    for (int i = 1; i < 18; ++i) {
        int off = 1 << (i - 1),
            end = icnt - (1 << i) + 1;
        for (int j = 1; j <= end; ++j)
            A[j][i] = choose(A[j][i - 1],
                A[j + off][i - 1]);
    }
}
int LUT[1024];
void buildLUT() {
    for (int i = 2; i < 1024; ++i)
        LUT[i] = LUT[i >> 1] + 1;
}
int ilg2(int n) {
    return n >> 10 ? LUT[n >> 10] + 10 : LUT[n & 1023];
}
void pre() {
    DFS(1);
    buildST();
    buildLUT();
}
int getLca(int u, int v) {
    int l = L[u], r = L[v];
    if (l > r) std::swap(l, r);
    int p = ilg2(r - l + 1);
    return choose(A[l][p], A[r - (1 << p) + 1][p]);
}
```

预处理 $O(n \lg n)$, 单次查询 $O(1)$。若非必要还是使用树剖做 LCA, 毕竟考场时间比较宝贵, 而且树剖表现不错, 空间消耗小。

11.1.4 Tarjan

当查询离线时, 可使用 Tarjan 算法。
代码如下:

Tarjan

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 500005;
struct G {
    struct Edge {
        int to, nxt;
    } E[size * 2];
    int last[size], cnt;
    G() : cnt(0) {}
    void addEdge(int u, int v) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u];
        last[u] = cnt;
    }
} g, qe;
struct Query {
    int ans, xorv;
} q[size];
int fa[size];
int find(int u) {
    return fa[u] == u ? u : fa[u] = find(fa[u]);
}
void DFS(int u, int p) {
    for(int i = g.last[u]; i; i = g.E[i].nxt) {
        int v = g.E[i].to;
        if(v != p) {
            DFS(v, u);
            fa[v] = u;
        }
    }
    for(int i = qe.last[u]; i; i = qe.E[i].nxt) {
        int id = qe.E[i].to;
        if(q[id].ans == 0x3f3f3f3f) {
            q[id].ans = find(q[id].xorv ^ u);
```

```
            } else
                q[id].ans = 0x3f3f3f3f;
        }
    }
    int main() {
        int n = read();
        int m = read();
        int s = read();
        for(int i = 1; i <= n; ++i)
            fa[i] = i;
        for(int i = 1; i < n; ++i) {
            int u = read();
            int v = read();
            g.addEdge(u, v);
            g.addEdge(v, u);
        }
        for(int i = 0; i < m; ++i) {
            int u = read();
            int v = read();
            qe.addEdge(u, i);
            qe.addEdge(v, i);
            q[i].xorv = u ^ v;
        }
        DFS(s, 0);
        for(int i = 0; i < m; ++i)
            printf("%d\n", q[i].ans);
        return 0;
    }
```

算法正确性证明:

- 若询问的两个节点在一条从根节点出发的链上, 则第二被访问的节点为深度较深的节点, 而深度较浅的节点的父亲仍然是自己;
- 若询问的两个节点在 LCA 的左右子树上, 则第二被访问的节点被访问时, 第一被访问节点的祖先已被置为 LCA。

优化 可以把 find 改为用队列实现, 迭代更新父亲。

11.2 链剖分

以下方法只适用于静态树。

11.2.1 轻重链剖分

对于每个节点, 选取其子树最大的儿子作为重儿子, 其余节点为轻儿子。然后将连续的重儿子串成一条链(DFS 序上连续), 每条链上最浅的节点为链头, 链内每个节点都指向链头方便跨越轻边跳跃到上一条链。

```
int d[size],p[size],siz[size],son[size];
void buildTree(int u) {
    siz[u]=1;
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(v!=p[u]) {
            p[v]=u;
            d[v]=d[u]+1;
            buildTree(v);
            siz[u]+=siz[v];
            if(siz[son[u]]<siz[v])
                son[u]=v;
        }
    }
}
int top[size];
void buildChain(int u) {
    if(son[u]) {
        top[son[u]]=top[u];
        buildChain(son[u]);
    }
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(!top[v]) {
            top[v]=v;
            buildChain(v);
        }
    }
}
```

注意调用 buildChain 前要先令 $top[1]=1$。

性质 11.1 剖分后的任意点到根经过的轻边与重链的数目为 $O(\lg n)$。

证明:

- 由于每个轻儿子的大小不超过父亲大小的一半, 所以每次经过一次轻边时, 所在子树大小至少增加一倍。因此经过 $O(\lg n)$ 条轻边。
- 因为重链由轻边连接, 所以重链数 $=$ 轻边数 $+1$, 因此经过重链数也是 $O(\lg n)$。

11.2.1.1 在链上统计中的应用

使用线段树维护信息需要保证被操作链的节点尽可能连续排列, 使用轻重链剖分后的 DFS 序保证了重链的点连续且修改重链数为 $O(\lg n)$, 搭配线段树可在 $O(\lg^2 n)$ 内单次链上修改/查询。

模板与求 LCA 类似:

```
int top[size],id[size],pid[size],icnt=0;
void buildChain(int u) {
    id[u]=++icnt;
    pid[icnt]=u;//for build
```

```
    //...
}
void build(int l,int r,int id) {
    if(l==r) {
        int u=pid[l];
        //...
    }
    //...
}
typedef void (*Func)(int,int,int);
template<Func func>
void applyImpl(int l,int r) {
    nl=l,nr=r;
    func(1,icnt,1);
}
template<Func func>
void apply(int u,int v) {
    while(top[u]!=top[v]) {
        if(d[top[u]]<d[top[v]])
            std::swap(u,v);
        applyImpl<Func>(id[top[u]],id[u]);
        u=p[top[u]];
    }
    if(d[u]>d[v])
        std::swap(u,v);
    applyImpl<Func>(id[u],id[v]);
}
```

11.2.2 长链剖分

顾名思义就是把子树深度最深的儿子当做重儿子进行剖分。

11.2.2.1 快速合并以深度为下标的信息

令同一条链上的点共享一块 dp 存储区, 统计当前节点信息时, 直接继承重儿子的信息, 然后暴力合并轻儿子的链的信息。

为了能够让父亲 $O(1)$ 继承重儿子的信息, 按照重儿子优先的 DFS 序分配 dp 数组起始位置。此时父亲的 dp 起点恰好在重儿子 dp 起点的前一位, 满足深度上的相对关系, 因此信息可以 $O(1)$ 继承。空间复杂度也为 $O(n)$。

复杂度证明:注意每个轻儿子(链头)子树信息合并到长链上的复杂度是 O(轻儿子链长), 且链链不相交, 因此总复杂度 $O(n)$; 而父亲继承重儿子信息的复杂度为 $O(1)$, 总复杂度 $O(n)$。所以最终复杂度为 $O(n)$。

血泪史:CF1009F Dominant Indices

要注意树退化成一条链的情况,对每个点 O(最长链长) 扫一遍会退化为 $O(n^2)$,考虑同时继承重儿子的答案,以次长链长(最长轻儿子链长)为更新长度扫描更新答案。也就是说重儿子信息必须 $O(1)$ 处理。

```
#include <cstdio>
int read() {
    int res = 0, c;
```

```
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
void print(int x) {
    if(x >= 10)
        print(x / 10);
    putchar('0' + x % 10);
}
const int size = 1000005;
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int son[size], cl[size];
int DFS(int u, int p, int d) {
    int md = ++d;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p) {
            int vmd = DFS(v, u, d);
            if(vmd > md)
                md = vmd, son[u] = v;
        }
    }
    cl[u] = md - d + 1;
    return md;
}
int dp[size], ans[size], alp = 0;
bool vis[size];
int solve(int u) {
    int beg = alp++;
    ++dp[beg];
    vis[u] = true;
    int res = beg, sp = beg, cmd = 0;
    if(son[u])
        sp = solve(son[u]);
    int scl = beg + 1;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
```

```
        if(!vis[v]) {
            int lp = beg + 1, rp = alp,
                end = rp + cl[v];
            solve(v);
            while(rp < end)
                dp[lp++] += dp[rp++];
            if(scl < lp)
                scl = lp;
        }
    }
    for(int i = beg; i < scl; ++i)
        if(dp[i] > cmd)
            res = i, cmd = dp[i];
    if(dp[sp] > cmd || (dp[sp] >= cmd && sp < res))
        res = sp;
    ans[u] = res - beg;
    return res;
}
int main() {
    int n = read();
    for(int i = 1; i < n; ++i) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    DFS(1, 0, 0);
    solve(1);
    for(int i = 1; i <= n; ++i) {
        print(ans[i]);
        putchar('\n');
    }
    return 0;
}
```

11.2.2.2 快速求 k 级祖先

预处理

1. 使用树上倍增 $O(n \lg n)$ 预处理第 2^k 级祖先。
2. 对每条长链预处理长链顶的前 “链长” 个祖先, 以及链上的所有点的编号。

查询

定理 11.2 任意一点的 k 级祖先所在的链长一定大于等于 k。

证明:

- 若该点与 k 级祖先在同一长链, 显然定理成立。
- 否则, k 级祖先所在长链的叶节点肯定不比自己浅, 定理同样成立。

步骤如下:

1. 用倍增数组跳 k 的最高位, 设剩余层数为 k', 有 $k' < \frac{k}{2}$;
2. 由定理 11.2得当前节点所在链的链长严格大于 k', 利用链头向上/向下的数组 $O(1)$ 查询。

预处理 $O(n \lg n)$, 查询 $O(1)$。
注意取最高位的复杂度要为 $O(1)$,参见 19.2.4节。
长链剖分参考了 MoebiusMeow 的博客[1] 与后缀自动机·张的文章[2]。

11.3 树的直径

11.3.1 定义与计算

树的直径就是树的最长链。

- 求直径长: 一遍 DFS 对于每个点计算其到子树的最大与次大距离, 然后相加更新答案。
- 求某条直径:
  1. 任选一点 a, DFS 计算与其相距最远的点 b;
  2. 从 b 点开始 DFS 计算与其相距最远的点 c;
  3. $b - c$ 就是树的直径。

注意此法不能用于带负权直径。

11.3.2 性质

性质 11.3 距树上任意一点最远的点必定在树的直径上。

性质 11.4 所有直径的中心(点或边)相同,树上任意一点到最远点的路径必经过这个中心。

性质 11.5 所有直径的偏心距相等。

下面这个性质可用于计算两棵子树合并后的直径:

性质 11.6 树的直径端点是子树的直径端点。

解与树上点到路径距离有关的 dp 题时一般先考虑求直径, 然后双指针法 dp。
上述性质参考了里阿奴摩西的博客[3]。

11.3.3 多子树直径查询

将原树按 DFS 序展开到线段树上, 对于每个区间维护其直径与两端点, 合并时取四个点中最远点对当做新树的直径端点。由于询问保证区间内的点连通, 不必考虑合并时区间内的点是否连通。

注意用 LCA 求距离时要使用欧拉序 +ST 表法。
预处理 $O(n \lg n)$, 查询 $O(\lg n)$。
上述内容参考了 rzO_KQP_Orz 的博客[4]。

[1] 长链剖分随想 - MoebiusMeow https://www.cnblogs.com/meowww/p/6403515.html
[2] 长链剖分之 O(nlgn)-O(1) 求 k 级祖先 https://zhuanlan.zhihu.com/p/25984772
[3] [树的直径] BZOJ 1999 [Noip2007]Core 树网的核 https://blog.csdn.net/u014609452/article/details/69351006
[4] 用线段树维护树的直径 https://blog.csdn.net/rzo_kqp_orz/article/details/52280811

11.4 Dsu On Tree

Dsu On Tree 用于求解**静态树**的**子树统计**问题。主要思路是在重链剖分后，令父亲直接继承重儿子的信息，暴力枚举轻儿子子树节点更新信息。

步骤如下：

1. DFS 轻儿子计算轻儿子子树节点的子树信息，统计完毕后消去影响；
2. DFS 重儿子计算重儿子子树节点的子树信息，统计完毕后保留影响；
3. 暴力枚举轻儿子子树节点更新信息；
4. 计算当前节点自身的影响；
5. 此时维护的信息为该节点的子树信息，记录答案；
6. 若标记为消去影响，则重置维护的信息（**注意清空数组的复杂度不能高于其余部分的复杂度**）。

Dsu On Tree

```
void DFS(int u, int p, bool erase) {
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p && v != son[u])
            DFS(v, u, true);
    }
    if(son[u])
        DFS(son[u], u, false);
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p && v != son[u]) {
            for(int j = L[v]; j <= R[v]; ++j)
                add(idc[j]);//idc数组为已在DFS序上展开的数据
        }
    }
    add(c[u]);
    //记录信息
    if(erase) {
        for(int i = L[u]; i <= R[u]; ++i)
            erase(idc[i]);//为了保证复杂度按需清空
        //重置其余维护信息
    }
}
```

若可以 $O(1)$ 计算加上/删除一个点的影响，则时间复杂度为 $O(n \lg n)$。

证明：因为每个节点到根经过轻边条数为 $O(\lg n)$，所以每个节点被作为轻儿子的子树节点合并的次数为 $O(\lg n)$，总时间复杂度为 $O(n \lg n)$。

以上内容参考了 Arpa 的博客[5]。

[5] [Tutorial] Sack (dsu on tree) - Codeforces http://codeforces.com/blog/entry/44351

11.5 Purfer Sequence

Purfer Sequence 用来表示树的结构(**带标号**)。

11.5.1 构造 Purfer Sequence

1. 在当前树中找到标号最小的叶子节点;
2. 向序列加入与该叶子节点相连的节点标号, 并将该叶子节点删除。
3. 重复步骤 1 直至只剩 2 个节点。

由此可得两个性质:

性质 11.7 (唯一性) 一个大小为 n 的树对应一个长度为 $n-2$ 的 *Purfer Sequence*。

性质 11.8 一个度数为 d 的节点在 *Purfer Sequence* 中出现 $d-1$ 次。

11.5.2 恢复原树

1. 统计节点在 Purfer Sequence 中出现的次数得到每个点的度数, 记为 $d[u]$;
2. 对于序列中的每一个编号 v, 选取 $d[u]=1$ 且标号最小的节点 u, 连接 (u,v), 然后 $--d[u], --d[v]$(实际上只需用平衡树维护度数为 1 的节点, 在平衡树上删除点 u, $--d[v]$ 后判断是否要插入点 v);
3. 将剩下两个 $d[u]=1$ 的点连边。

11.5.3 计数应用

可根据度数与出现次数的关系计算满足度数要求的树的数目。**注意无解和 $n \le 2$ 时的情况。**

以上内容参考了 JMJST 的博客[6]。

11.6 虚树

在多次询问的树形 dp 问题中, 若遇到询问总点数与树的大小同数量级的情况, 可以在每次询问中将询问节点建成一棵 ‘‘虚树’’, 然后对虚树做树形 dp。这样做可以有效地降低 dp 规模($qT(n) \to \sum_{i=1}^{q}(O(m_i \lg m_i)+T(O(2m_i)))$)。

11.6.0.1 构造过程

首先预处理从根节点 DFS 遍历整棵树, 记录 DFS 序与深度, 预处理计算 LCA 需要的信息, 同时维护其他信息。

对于每一次询问:

1. 将询问节点按 DFS 序排序;
2. 将第一个节点加入栈中, 栈上维护的是当前节点到根的链;

[6]BZOJ 1005 [HNOI2008] 明明的烦恼(组合数学 Purfer Sequence)- jianzhang.zj http://www.cnblogs.com/zhj5chengfeng/archive/2013/08/23/3278557.html

3. 对于剩下每一个节点 u:
   (a) 计算自己与栈顶节点 v 的 lca;
   (b) 若栈中第二个节点 p 的深度比 lca 大, 则连接 $p \to v$, 弹出 v。重复该步骤直至不满足条件;
   (c) 若 lca 比 v 浅, 连接 $lca \to v$, 弹出 v。
   (d) 若栈为空或 v 比 lca 浅, 加入节点 lca。
   (e) 加入节点 u。
4. 此时栈上还有一条链, 将链加入树中, 栈底就是根节点。

11.6.0.2 算法解释

lca 有两种可能:

- lca 为 v:此时的操作只有加入节点 u, 就是简单地将自己挂在虚树中的父亲下。
- u 和 v 在 lca 的两棵子树下: 首先不断地跳 v 直至 u 和 v 在虚树上的直接祖先为 lca, 然后记此时栈顶为 p, 继续分类:
  - p 为 lca, 连接 $lca \to v$ 后把 v 换成 u。
  - p 为 lca 的祖先, 连接 $lca \to v$ 后把 v 换成 lca 与 u。
  - lca 为原链头的祖先(对应空栈), 把链全部折叠后加入节点 lca 与 u。

11.6.0.3 算法实现

```
int buildTree(int k) {
    g2.cnt = 0;
    int top = 1;
    std::sort(id + 1, id + k + 1, cmp);
    st[1] = id[1];
    for (int i = 2; i <= k; ++i) {
        int u = id[i];
        int lca = getLca(u, st[top]);
        while (top > 1 && d[lca] < d[st[top - 1]]) {
            g2.addEdge(st[top - 1], st[top]);
            --top;
        }
        if (d[lca] < d[st[top]]) {
            g2.addEdge(lca, st[top]);
            --top;
        }
        if (top == 0 || d[st[top]] < d[lca])
            st[++top] = lca;
        st[++top] = u;
    }
    while (top > 1) {
        g2.addEdge(st[top - 1], st[top]);
        --top;
```

```
    }
    return st[1];
}
```

虚树加边不必存边权, 在 DFS 预处理时处理到根的权值和, 需要边权时两点值相减可得。

注意有时需要把根节点强制加入到虚树中去。

警告:每次 dp 的清空复杂度要与询问点数一致。

11.7 点分治

点分治就是每次选择树的重心(儿子的子树大小的最大值最小的节点)作为分治点, 以分治点为根将整棵树分为多棵子树, 统计经过当前节点的路径贡献, 然后递归每棵子树分治。这种方法一般用来解决**路径统计问题**。

能够使用树形 DP 解决(路径信息可合并)的不要使用点分治。

11.7.1 静态点分治

11.7.1.1 重心性质

性质 11.9 重心的儿子子树大小不超过整棵树大小的一半。

证明:考虑重心为 R, 设 R 的某个儿子 u 满足其子树大小超过 $\frac{N}{2}$, 那么点 u 比点 R 更优, 与重心为 R 的条件矛盾。

性质 11.10 所有点到重心的距离和最小,到两个重心的距离和相等(若树大小为奇数则只有一个重心)。

性质 11.11 两棵树合并后的重心在这两棵树的重心的路径上。

性质 11.12 添加或减少一个叶子节点,重心最多偏移一条边。

这个性质搭配启发式合并可以用来快速处理新连通块的重心。

例题 LuoguP4299 首都:

根据性质 11.10可知该题要维护的是动态连通块的重心, 且重心一定在这两棵树的重心的路径上。那么可以在其中一棵树上从连接点开始逐步长叶子, 利用性质 11.12, 每次试探是否要往另一连通块的重心移动。使用启发式合并可以保证合并复杂度, 至于快速查询距离和/ 子树大小, 可以使用 LCT 维护。时间复杂度 $O(n \lg^2 n)$。

查询首都时应该使用常数更小的并查集维护, 向某个节点的移动可以使用树链剖分支持查询(按照是否在到根的链上分类, 结合 14.5.2.3中提到的 jump 函数询问)。移动目标仅需指定为较小子树的连接点, 因为再移动就会违反性质 11.9。维护最大子树大小比维护距离和更方便, 不过由于要维护可删堆, 性能稍差一点。

参考代码($O(n \lg^2 n)$):

```
0 #include <algorithm>
  #include <cstdio>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
```

```
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
int getOp() {
    int c;
    do
        c = getchar();
    while(c < 'A' || c > 'Z');
    return c;
}
const int size = 100005;
typedef long long Int64;
#define asInt64 static_cast<Int64>
struct Node {
    int p, c[2], siz, rsiz, icnt;
    Int64 lsum, rsum, isum;
    bool rev;
} T[size];
#define ls T[u].c[0]
#define rs T[u].c[1]
bool isRoot(int u) {
    int p = T[u].p;
    return T[p].c[0] != u && T[p].c[1] != u;
}
int getPos(int u) {
    int p = T[u].p;
    return T[p].c[1] == u;
}
void connect(int u, int p, int c) {
    T[u].p = p;
    T[p].c[c] = u;
}
void pushDown(int u) {
    if(T[u].rev) {
        std::swap(ls, rs);
        std::swap(T[u].lsum, T[u].rsum);
        T[ls].rev ^= 1;
        T[rs].rev ^= 1;
        T[u].rev = false;
    }
}
void update(int u) {
    pushDown(ls);
    pushDown(rs);
    T[u].siz = T[ls].siz + T[rs].siz + 1;
    T[u].rsiz =
        T[ls].rsiz + T[rs].rsiz + 1 + T[u].icnt;
```

```
    T[u].lsum = T[u].isum + T[ls].lsum + T[rs].lsum +
        (T[ls].siz + 1) *
            asInt64(T[u].icnt + T[rs].rsiz) +
        T[ls].siz;
    T[u].rsum = T[u].isum + T[ls].rsum + T[rs].rsum +
        (T[rs].siz + 1) *
            asInt64(T[u].icnt + T[ls].rsiz) +
        T[rs].siz;
}
void rotate(int u) {
    int ku = getPos(u);
    int p = T[u].p;
    int kp = getPos(p);
    int pp = T[p].p;
    int t = T[u].c[ku ^ 1];
    T[u].p = pp;
    if(!isRoot(p))
        T[pp].c[kp] = u;
    connect(p, u, ku ^ 1);
    connect(t, p, ku);
    update(p);
    update(u);
}
void push(int u) {
    if(!isRoot(u))
        push(T[u].p);
    pushDown(u);
}
void splay(int u) {
    push(u);
    while(!isRoot(u)) {
        int p = T[u].p;
        if(!isRoot(p))
            rotate(getPos(u) == getPos(p) ? p : u);
        rotate(u);
    }
}
void access(int u) {
    int v = 0;
    do {
        splay(u);
        if(rs) {
            pushDown(rs);
            T[u].icnt += T[rs].rsiz;
            T[u].isum += T[rs].lsum;
        }
        rs = v;
        if(rs) {
            T[u].icnt -= T[rs].rsiz;
            T[u].isum -= T[rs].lsum;
```

```
            }
            update(u);
            v = u;
            u = T[u].p;
        } while(u);
    }
    void makeRoot(int u) {
        access(u);
        splay(u);
        T[u].rev ^= 1;
        pushDown(u);
    }
    void link(int u, int v) {
        makeRoot(u);
        access(v);
        splay(v);
        T[u].p = v;
        T[v].icnt += T[u].rsiz;
        T[v].isum += T[u].lsum;
        update(v);
    }
    Int64 query(int u) {
        access(u);
        splay(u);
        return T[u].rsum;
    }
    struct G {
        struct Edge {
            int to, nxt;
        } E[size * 2];
        int last[size], cnt;
        G() : cnt(0) {}
        void addEdge(int u, int v) {
            ++cnt;
            E[cnt].to = v, E[cnt].nxt = last[u];
            last[u] = cnt;
        }
    } g1, g2;
    int p[size], son[size], d[size];
    int buildTree(int u) {
        int siz = 1, msiz = 0;
        for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
            int v = g1.E[i].to;
            if(v != p[u]) {
                d[v] = d[u] + 1;
                p[v] = u;
                int vsiz = buildTree(v);
                siz += vsiz;
                if(vsiz > msiz)
                    son[u] = v, msiz = vsiz;
```

```
            }
        }
        return siz;
}
int top[size];
void buildChain(int u) {
    if(son[u]) {
        top[son[u]] = top[u];
        buildChain(son[u]);
    }
    for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
        int v = g1.E[i].to;
        if(!top[v]) {
            top[v] = v;
            buildChain(v);
        }
    }
}
int lca(int u, int v) {
    while(top[u] != top[v]) {
        if(d[top[u]] < d[top[v]])
            std::swap(u, v);
        u = p[top[u]];
    }
    return d[u] < d[v] ? u : v;
}
int jump(int u, int lca) {
    int res = -1;
    while(top[u] != top[lca]) {
        res = top[u];
        u = p[top[u]];
    }
    return u == lca ? res : son[lca];
}
int nxt(int u, int v) {
    if(lca(u, v) == u)
        return jump(v, u);
    else
        return p[u];
}
int crt, nrt, dst;
void clear(int u) {
    T[u].p = ls = rs = T[u].icnt = T[u].isum =
        T[u].lsum = T[u].rsum = T[u].rev = 0;
    T[u].siz = T[u].rsiz = 1;
}
void DFSAdd(int u, int p) {
    clear(u);
    link(u, p);
    if(dst != -1) {
```

```
            Int64 vo = query(crt), vc = query(nrt);
            if(vo > vc || (vo == vc && nrt < crt)) {
                crt = nrt;
                if(crt == dst)
                    dst = -1;
                else
                    nrt = nxt(crt, dst);
            }
        }
        for(int i = g2.last[u]; i; i = g2.E[i].nxt) {
            int v = g2.E[i].to;
            if(v != p)
                DFSAdd(v, u);
        }
    }
    struct Op {
        int op, u, v;
    } A[size * 2];
    int fa[size], rk[size], siz[size], rt[size];
    int find(int x) {
        return fa[x] ? fa[x] = find(fa[x]) : x;
    }
    int merge(int u, int v) {
        if(rk[u] < rk[v]) {
            fa[u] = v;
            return v;
        } else {
            fa[v] = u;
            if(rk[u] == rk[v])
                ++rk[u];
            return u;
        }
    }
    int main() {
        int n = read();
        int m = read();
        int xorv = 0;
        for(int i = 1; i <= n; ++i) {
            clear(i);
            siz[i] = 1;
            rt[i] = i;
            xorv ^= i;
        }
        for(int i = 1; i <= m; ++i) {
            A[i].op = getOp();
            switch(A[i].op) {
                case 'A': {
                    A[i].u = read();
                    A[i].v = read();
                    g1.addEdge(A[i].u, A[i].v);
```

```
                g1.addEdge(A[i].v, A[i].u);
            } break;
            case 'Q': {
                A[i].u = read();
            } break;
        }
    }
    for(int i = 1; i <= n; ++i) {
        if(!p[i]) {
            buildTree(i);
            top[i] = i;
            buildChain(i);
        }
    }
    for(int i = 1; i <= m; ++i) {
        switch(A[i].op) {
            case 'A': {
                int fu = find(A[i].u),
                    fv = find(A[i].v);
                xorv ^= rt[fu] ^ rt[fv];
                if(siz[fu] < siz[fv]) {
                    std::swap(A[i].u, A[i].v);
                    std::swap(fu, fv);
                }
                crt = rt[fu], nrt = nxt(crt, A[i].v),
                dst = A[i].v;
                g2.addEdge(A[i].u, A[i].v);
                g2.addEdge(A[i].v, A[i].u);
                DFSAdd(A[i].v, A[i].u);
                int cur = merge(fu, fv);
                siz[cur] = siz[fu] + siz[fv];
                rt[cur] = crt;
                xorv ^= crt;
            } break;
            case 'Q':
                printf("%d\n", rt[find(A[i].u)]);
                break;
            case 'X':
                printf("%d\n", xorv);
                break;
        }
    }
    return 0;
}
```

当然根据性质 11.11, 可以直接 link 两棵树, 然后 split 出两个重心的链, 根据性质 11.9 判重心, 在链上 DFS 剪枝查询(每次都往子树较大的儿子走), 复杂度由 splay 保证。由于去掉了离线建图支持定向移动的部分且只需要维护子树大小, 代码量小很多。由于链两端都是重心, 查询时不必考虑最大虚子树, 因为它们不可能超过整棵树大小的一半。

该方法参考了 FlashHu 的题解[7]。
参考代码($O(n \lg n)$):

```
#include <algorithm>
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
int getOp() {
    int c;
    do
        c = getchar();
    while(c < 'A' || c > 'Z');
    return c;
}
const int size = 100005;
struct Node {
    int p, c[2], rsiz, icnt;
    bool rev;
} T[size];
#define ls T[u].c[0]
#define rs T[u].c[1]
bool isRoot(int u) {
    int p = T[u].p;
    return T[p].c[0] != u && T[p].c[1] != u;
}
int getPos(int u) {
    int p = T[u].p;
    return T[p].c[1] == u;
}
void connect(int u, int p, int c) {
    T[u].p = p;
    T[p].c[c] = u;
}
void pushDown(int u) {
    if(T[u].rev) {
        std::swap(ls, rs);
        T[ls].rev ^= 1;
        T[rs].rev ^= 1;
        T[u].rev = false;
```

[7]题解 P4299【首都】
https://www.luogu.org/blog/flashblog/solution-p4299

```
    }
}
void update(int u) {
    T[u].rsiz =
        T[ls].rsiz + T[rs].rsiz + 1 + T[u].icnt;
}
void rotate(int u) {
    int ku = getPos(u);
    int p = T[u].p;
    int kp = getPos(p);
    int pp = T[p].p;
    int t = T[u].c[ku ^ 1];
    T[u].p = pp;
    if(!isRoot(p))
        T[pp].c[kp] = u;
    connect(p, u, ku ^ 1);
    connect(t, p, ku);
    update(p);
    update(u);
}
void push(int u) {
    if(!isRoot(u))
        push(T[u].p);
    pushDown(u);
}
void splay(int u) {
    push(u);
    while(!isRoot(u)) {
        int p = T[u].p;
        if(!isRoot(p))
            rotate(getPos(u) == getPos(p) ? p : u);
        rotate(u);
    }
}
void access(int u) {
    int v = 0;
    do {
        splay(u);
        if(rs)
            T[u].icnt += T[rs].rsiz;
        rs = v;
        if(rs)
            T[u].icnt -= T[rs].rsiz;
        update(u);
        v = u;
        u = T[u].p;
    } while(u);
}
void makeRoot(int u) {
    access(u);
```

```
    splay(u);
    T[u].rev ^= 1;
    pushDown(u);
}
void split(int u, int v) {
    makeRoot(u);
    access(v);
    splay(v);
}
void link(int u, int v) {
    split(u, v);
    T[u].p = v;
    T[v].icnt += T[u].rsiz;
    update(v);
}
int fa[size], rk[size], rt[size];
int find(int x) {
    return fa[x] ? fa[x] = find(fa[x]) : x;
}
int merge(int u, int v) {
    if(rk[u] < rk[v]) {
        fa[u] = v;
        return v;
    } else {
        fa[v] = u;
        if(rk[u] == rk[v])
            ++rk[u];
        return u;
    }
}
int DFSCmp(int u) {
    int tot = T[u].rsiz, half = tot / 2, lsiz = 0,
        rsiz = 0, crt = 1 << 30;
    while(u && lsiz <= half && rsiz <= half) {
        pushDown(u);
        int cl = lsiz + T[ls].rsiz,
            cr = rsiz + T[rs].rsiz;
        if(cl <= half && cr <= half) {
            if(tot & 1) {
                crt = u;
                break;
            } else
                crt = std::min(crt, u);
        }
        if(cl < cr) {
            lsiz += T[ls].rsiz + T[u].icnt + 1;
            u = rs;
        } else {
            rsiz += T[rs].rsiz + T[u].icnt + 1;
            u = ls;
```

```
        }
    }
    splay(crt);
    return crt;
}
int main() {
    int n = read();
    int m = read();
    int xorv = 0;
    for(int i = 1; i <= n; ++i) {
        T[i].rsiz = 1;
        rt[i] = i;
        xorv ^= i;
    }
    for(int i = 1; i <= m; ++i) {
        switch(getOp()) {
            case 'A': {
                int u = read();
                int v = read();
                int fu = find(u), fv = find(v);
                xorv ^= rt[fu] ^ rt[fv];
                link(u, v);
                split(rt[fu], rt[fv]);
                int cur = merge(fu, fv);
                rt[cur] = DFSCmp(rt[fv]);
                xorv ^= rt[cur];
            } break;
            case 'Q':
                printf("%d\n", rt[find(read())]);
                break;
            case 'X':
                printf("%d\n", xorv);
                break;
        }
    }
    return 0;
}
```

树的同构判定 对于一棵有根树可以使用一些 Hash 规则计算子树的 Hash 值(合并子树 Hash 值的运算需要满足交换律或排序后合并, 我的做法是计算多个满足交换律的运算合并值, 然后把这些值胡乱混合)。以无根树的重心(1-2 个)为根, 就可以计算整棵树的 Hash 值, 用于树的同构判定。

血泪史:存储当前最优点的数组大小要开 n 而不是 2。尽管重心不超过 2 个,但在处理过程中仍然存在超过 2 个的现行最优点。我在下面的板子中由于 kp 数组开小了而导致 last 数组被覆写产生环,因此我调了一晚上。。。 望引以为戒。

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <map>
```

```
const int size = 55;
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int kp[size], kcnt, cmsiz, n;
int pre(int u, int p) {
    int siz = 1, mx = 0;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p) {
            int sv = pre(v, u);
            mx = std::max(mx, sv);
            siz += sv;
        }
    }
    mx = std::max(n - siz, mx);
    if(mx < cmsiz) {
        kcnt = 0;
        cmsiz = mx;
    }
    if(mx == cmsiz)
        kp[kcnt++] = u;
    return siz;
}
int DFS(int u, int p) {
    int a = 1, b = 1;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p) {
            int hv = DFS(v, u);
            a += hv, b *= hv;
        }
    }
    return a * b + a + b + (a ^ b);
}
int getHash() {
    cnt = 0;
    memset(last, 0, sizeof(last));
    scanf("%d", &n);
    int rt;
    for(int i = 1; i <= n; ++i) {
        int p;
        scanf("%d", &p);
        if(p) {
```

```
            addEdge(p, i);
            addEdge(i, p);
        } else
            rt = i;
    }
    kcnt = 0, cmsiz = 1 << 30;
    pre(rt, 0);
    return kcnt == 1 ?
        DFS(kp[0], 0) :
        std::min(DFS(kp[0], 0), DFS(kp[1], 0));
}
std::map<int, int> hashTable;
int main() {
    int m;
    scanf("%d", &m);
    for(int i = 1; i <= m; ++i) {
        int& hid = hashTable[getHash()];
        if(!hid)
            hid = i;
        printf("%d\n", hid);
    }
    return 0;
}
```

11.7.1.2 重心选择

以当前连通块内任意一点为根(当然是上一个分治点的儿子)DFS, 计算每个节点儿子的子树大小的最大值, 然后与除自己子树外的节点数求最大值(儿子为有根树意义下的父亲时的子树), 就得到当前节点的权重。

使用 vis 数组来标记节点是否已成为分治点, 阻止 DFS 过程跨出连通块。

getRoot

```
bool vis[size];
int root,tsiz,msiz,siz[size];
void getRootImpl(int u,int p) {
    int maxs=0;
    siz[u]=1;
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(!vis[v] && v!=p) {
            getRootImpl(v,u);
            siz[u]+=siz[v];
            maxs=std::max(maxs,siz[v]);
        }
    }
    maxs=std::max(maxs,tsiz-siz[u]);
    if(maxs<msiz) {
        msiz=maxs;
        root=u;
```

```
    }
}
int getRoot(int u,int csiz) {
    msiz=1<<30;
    tsiz=csiz;
    getRootImpl(u,0);
    return root;
}
```

11.7.1.3 分治与全局统计

每次把重心作为分治点, DFS 统计从分治点出发的路径贡献, 将分治点到子树各点的路径信息平铺成序列, 用 $O(n)$(单调队列、双指针)或 $O(n \lg n)$(线段树)算法两两合并, 再对于每棵子树去除来自同一棵子树的贡献(因为这些路径经过了两次从分治点到子树的边, 不是简单路径)。

divide

```
void divide(int u) {
    //count u->child
    vis[u]=true;
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(!vis[v]) {
            //minus v->u->v
            divide(getRoot(v,siz[v]));
        }
    }
}
```

Warning:理论上调用 $getRoot$ 时需要重新 DFS 一遍算出所有子连通块的 siz,但是这里直接使用上一次 DFS 的 $siz[v]$ 代替。事实上对于点分树上的儿子在实际树中夹在父亲和爷爷中间的情况,$siz[v]$ 并不是真正的子树大小,但是实践表明这个简化方法的时间复杂度并不会退化,且层数保证严格 $O(\lg n)$,可以放心使用内存池分配空间。不过要用到 siz 时仍需要重新 DFS 一遍。时间复杂度证明详见 LCA 的博客[8]。

统计过程也可以使用树形 dp 的惯用技巧: 在统计从分治点出发的贡献时将儿子子树分开处理, 每棵子树 DFS 一遍查询与已处理子树节点(或者与分治点本身)的贡献, 再 DFS 一遍维护数据结构(线段树常数好大)便于下一次查询。如此就不会统计来自相同子树的贡献了, 注意数据结构的清零要控制在子树规模内。**关键:利用时间来存储数据。**两趟 DFS 太慢, 可以一趟 DFS 将信息平铺到序列上, 然后查询 + 加入数据结构。

11.7.1.4 局部统计

有时需要计算从每个点出发的所有路径信息(最好具有无向性与可合并性), 同样可以使用点分治处理。从点 u 出发的路径, 它要么在将点 u 作为分治点时被 DFS, 要么在将点 u 在点分树上的祖先作为分治点时 DFS 到 u, 并将这一段路径与其它子树的路径合并时被统计到。

[8] 一种基于错误的寻找重心方法的点分治的复杂度分析 http://liu-cheng-ao.blog.uoj.ac/blog/2969

我原先统计合并路径的方法: 使用上文类似树形 DP 的方法, 不过由于是局部统计, 统计到点 u 时能够合并的路径只有合并之前被 DFS 的子树的路径。那么需要使用 vector 存边, 然后正反各做一遍。

从租酥雨的博客[9]学到了更简单的统计方法: 先 DFS 分治点的整个子树, 每次统计某个儿子的路径时, 先删掉该儿子的子树路径贡献, 统计, 再恢复儿子的子树路径贡献。这么做还可以顺便计算单点的贡献和从 u 出发的路径贡献。

11.7.1.5 时间复杂度

点分治会带来 $O(\lg n)$ 的复杂度, 证明:

根据性质 11.9, 点分治的层数为 $O(\lg n)$, 而且每层的总规模都是 n。

11.7.2 动态点分治

动态点分治就是在分治时连接当前分治点与子连通块的分治点, 这些点构成了一棵点分树, 多次查询时使用点分树来计算。

注意一般节点 u 需要维护**自己的子树节点到自己在点分树上父亲的信息**, 记为 A。那么节点 u 子树节点经过自身的路径信息就由儿子的信息 A 决定。修改信息时自底向上更新祖先的信息。由于层数控制在 $O(\lg n)$ 内, 更新祖先的复杂度有了保证。

11.7.2.1 例题

Luogu P4115 Qtree4[10]

首先点分治建出点分树。对于每个节点维护两个可修改堆, 堆 A 维护子树白节点到该节点父亲的距离, 堆 B 维护该节点儿子们的堆 A 最大值, 那么经过该节点的最长路径为堆 B 的最大值 + 堆 B 的次大值。注意自身就是白点的情况。再用一个全局可修改堆维护经过每个节点的最长路径。

实践中双 $std :: priority\_queue$ 维护可修改堆比 $std :: multiset$ 跑得更快, 参见第 3.8.3节。

代码如下:

Luogu P4115

```
0 #include <algorithm>
  #include <cstdio>
  #include <queue>
  int read() {
      int res = 0, c;
      bool flag = false;
      do {
          c = getchar();
          flag |= c == '-';
      } while(c < '0' || c > '9');
10    while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
          c = getchar();
      }
      return flag ? -res : res;
```

[9] [Luogu2664] 树上游戏 https://www.cnblogs.com/zhoushuyu/p/8311249.html

[10] 【P4115】Qtree4 - 洛谷 https://www.luogu.org/problemnew/show/P4115

```
}
int getOp() {
    int c;
    do
        c = getchar();
    while(c != 'C' && c != 'A');
    return c;
}
const int size = 100005;
struct Edge {
    int to, nxt, w;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].w = w;
    last[u] = cnt;
}
int icnt = 0, A[18][size * 2], L[size], d[size],
    len[size];
void pre(int u, int p) {
    A[0][++icnt] = u;
    L[u] = icnt;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p) {
            d[v] = d[u] + 1;
            len[v] = len[u] + E[i].w;
            pre(v, u);
            A[0][++icnt] = u;
        }
    }
}
int choose(int u, int v) {
    return d[u] < d[v] ? u : v;
}
void buildST() {
    for(int i = 1; i < 18; ++i) {
        int off = 1 << (i - 1),
            end = icnt - (1 << i) + 1;
        for(int j = 1; j <= end; ++j)
            A[i][j] =
                choose(A[i - 1][j], A[i - 1][j + off]);
    }
}
int ilg2(int x) {
    return 31 - __builtin_clz(x);
}
int getLca(int u, int v) {
    int l = L[u], r = L[v];
```

```
    if(l > r)
        std::swap(l, r);
    int p = ilg2(r - l + 1);
    return choose(A[p][l], A[p][r - (1 << p) + 1]);
}
int getDis(int u, int v) {
    return len[u] + len[v] - (len[getLca(u, v)] << 1);
}
bool vis[size];
int root, tsiz, msiz, siz[size];
void getRootImpl(int u, int p) {
    int maxs = 0;
    siz[u] = 1;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(!vis[v] && v != p) {
            getRootImpl(v, u);
            siz[u] += siz[v];
            maxs = std::max(maxs, siz[v]);
        }
    }
    maxs = std::max(maxs, tsiz - siz[u]);
    if(maxs < msiz) {
        msiz = maxs;
        root = u;
    }
}
int getRoot(int u, int csiz) {
    msiz = 1 << 30, tsiz = csiz;
    getRootImpl(u, 0);
    return root;
}
const int nan = 0xc0c0c0c0;
class Heap {
private:
    std::priority_queue<int> A, B;
    void update() {
        while(B.size() && B.top() == A.top())
            A.pop(), B.pop();
    }

public:
    void insert(int val) {
        if(val != nan)
            A.push(val);
    }
    void erase(int val) {
        if(val != nan)
            B.push(val);
    }
```

```
        void modify(int val, bool op) {
            if(val != nan)
                (op ? A : B).push(val);
        }
        int getMax() {
            update();
            return A.size() ? A.top() : nan;
        }
        int getMix() {
            int vala = getMax();
            if(vala != nan) {
                A.pop();
                int valb = getMax();
                A.push(vala);
                if(valb != nan)
                    return vala + valb;
            }
            return nan;
        }
        void replace(int a, int b) {
            erase(a);
            insert(b);
        }
    } df[size], md[size], glo;
    int fa[size];
    void DFS(int u, int p, int a, int b) {
        df[a].insert(getDis(u, b));
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(!vis[v] && v != p)
                DFS(v, u, a, b);
        }
    }
    void divide(int u) {
        md[u].insert(0);
        md[u].insert(0);
        if(fa[u])
            DFS(u, 0, u, fa[u]);
        vis[u] = true;
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
            if(!vis[v]) {
                int nrt = getRoot(v, siz[v]);
                fa[nrt] = u;
                divide(nrt);
                md[u].insert(df[nrt].getMax());
            }
        }
    }
    void modify(int p, int u, bool op) {
```

```
        int pp = fa[p];
        int dis = getDis(pp, u);
        int old = df[p].getMax();
        df[p].modify(dis, op);
        int now = df[p].getMax();
170     if(old != now) {
            int omix = md[pp].getMix();
            md[pp].replace(old, now);
            int nmix = md[pp].getMix();
            if(omix != nmix)
                glo.replace(omix, nmix);
        }
    }
    bool col[size];
    int main() {
180     int n = read();
        for(int i = 1; i < n; ++i) {
            int u = read();
            int v = read();
            int w = read();
            addEdge(u, v, w);
            addEdge(v, u, w);
        }
        pre(1, 0);
        buildST();
190     int rt = getRoot(1, n);
        divide(rt);
        for(int i = 1; i <= n; ++i)
            glo.insert(md[i].getMix());
        int q = read();
        while(q--) {
            if(getOp() == 'C') {
                int u = read();
                bool op = col[u];
                col[u] ^= 1;
200             {
                    int omix = md[u].getMix();
                    md[u].modify(0, op);
                    md[u].modify(0, op);
                    int nmix = md[u].getMix();
                    if(omix != nmix)
                        glo.replace(omix, nmix);
                }
                int p = u;
                while(fa[p]) {
210                 modify(p, u, op);
                    p = fa[p];
                }
            } else {
                int res = glo.getMax();
```

```
            if(res != nan)
                printf("%d\n", res);
            else
                puts("They have disappeared.");
        }
    }
    return 0;
}
```

11.7.2.2 与动态 DP 的区别

动态点分治与动态 DP 都可以在 $O(\lg n)$ 的复杂度内单次更新, 但是它们的应用场合不同。

- 动态点分治擅长与路径统计有关的问题, 询问一般带有 “路径”“距离” 等关键字。
- 动态 DP 擅长与子树统计有关的问题, 单点 DP 比较简单。

11.7.2.3 与距离有关的动态点分治

例题~~血泪史~~: bzoj3730: 震波

计算每个分治点子树内的路径, 每个分治点的父亲到自己子树的路径(用于去重)。使用这两类信息支持询问。

有一些要注意的地方:

- 关键在于选用什么数据结构维护路径(要求单点加减, 前缀查询)。使用平衡树当然可以实现, 但是常数大, 编码与 Debug 难度大。如果使用树状数组, 注意到点分治对于深度并没有保证, 是否会 MLE 呢?

  仔细分析后发现这么做并不会 MLE: 注意每个分治点子树内的路径长度是连续的, 所以 BIT 不可能出现空位。在最极端情况下(BIT 上的位置不会被复用), 由于每个节点最多在 $O(\lg n)$ 棵分治点子树内, 因此空间复杂度为 $O(n \lg n)$。

- 修改点权时跳点分树容易遗漏, 干脆给每个节点都记录其被引用的数据结构的指针 + 位置。
- 线性预处理 BIT 时记得判尾部, 单独判 $tot == n$ 或者强制令 $A[tot + 1].k = -1$。
- 可能存在儿子被夹在父亲和爷爷之间的情况, 不能使用相对距离逐级减去。查询时使用的是**查询点到当前点的距离**, 不过去重用的 BIT 仍然是当前点的(表达意义相同)。详见代码中的 query 部分。这个问题耽误了我 2 个小时。。。

  Update: 查询自己到某个祖先的距离可以不使用额外的树剖, 在祖先 DFS 其子树时就可以支持距离计算, 按顺序放入 vector。

参考代码:

```
#include <algorithm>
#include <cstdio>
#include <vector>
int read() {
    int res = 0, c;
    do
        c = getchar();
```

```
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005;
struct BIT {
    std::vector<int> A;
    int n;
    void init(int siz) {
        A.resize(siz + 1);
        n = siz;
    }
    void modify(int x, int v) {
        while(x <= n) {
            A[x] += v;
            x += x & -x;
        }
    }
    int query(int x) {
        x = std::min(n, x);
        if(x <= 0)
            return 0;
        int res = 0;
        while(x) {
            res += A[x];
            x -= x & -x;
        }
        return res;
    }
} sumT[size], subT[size];
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int tsiz, msiz, crt, siz[size];
bool vis[size];
void getRootImpl(int u, int p) {
    siz[u] = 1;
    int csiz = 0;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p && !vis[v]) {
```

```
            getRootImpl(v, u);
            siz[u] += siz[v];
            csiz = std::max(csiz, siz[v]);
        }
    }
    csiz = std::max(csiz, tsiz - siz[u]);
    if(csiz < msiz)
        msiz = csiz, crt = u;
}
int getRoot(int u, int siz) {
    msiz = 1 << 30, tsiz = siz;
    getRootImpl(u, 0);
    return crt;
}
int w[size], tot;
struct Info {
    int k, u;
    Info() {}
    Info(int k, int u) : k(k), u(u) {}
    bool operator<(const Info& rhs) const {
        return k < rhs.k;
    }
} A[size];
void DFS(int u, int p, int k) {
    if(k)
        A[++tot] = Info(k, u);
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(p != v && !vis[v])
            DFS(v, u, k + 1);
    }
}
std::vector<std::pair<BIT*, int> > uq[size];
void solve(BIT& T, int u, int k) {
    tot = 0;
    DFS(u, 0, k);
    std::sort(A + 1, A + tot + 1);
    int siz = A[tot].k;
    A[tot + 1].k = -1;
    T.init(siz);
    for(int i = 1, csum = 0; i <= tot; ++i) {
        csum += w[A[i].u];
        uq[A[i].u].push_back(
            std::make_pair(&T, A[i].k));
        if(A[i].k != A[i + 1].k) {
            int p = A[i].k;
            T.A[p] += csum;
            int dst = p + (p & -p);
            if(dst <= siz)
                T.A[dst] += T.A[p];
```

```
            csum = 0;
        }
    }
}
int fa[size];
void divide(int u) {
    vis[u] = true;
    solve(sumT[u], u, 0);
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(!vis[v]) {
            int nrt = getRoot(v, siz[v]);
            fa[nrt] = u;
            solve(subT[nrt], v, 1);
            divide(nrt);
        }
    }
}
int p[size], d[size], son[size];
int buildTree(int u) {
    int siz = 1, msiz = 0;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p[u]) {
            p[v] = u;
            d[v] = d[u] + 1;
            int vsiz = buildTree(v);
            siz += vsiz;
            if(msiz < vsiz)
                msiz = vsiz, son[u] = v;
        }
    }
    return siz;
}
int top[size];
void buildChain(int u) {
    if(son[u]) {
        top[son[u]] = top[u];
        buildChain(son[u]);
    }
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(!top[v]) {
            top[v] = v;
            buildChain(v);
        }
    }
}
int getLCA(int u, int v) {
    while(top[u] != top[v]) {
```

```
            if(d[top[u]] < d[top[v]])
                std::swap(u, v);
            u = p[top[u]];
        }
        return d[u] < d[v] ? u : v;
    }
    int dis(int u, int v) {
        return d[u] + d[v] - 2 * d[getLCA(u, v)];
    }
    int src, k;
    int query(int x, int rk) {
        int res = rk >= 0 ? w[x] + sumT[x].query(rk) : 0;
        if(fa[x]) {
            int d = dis(fa[x], src);
            res += query(fa[x], k - d);
            res -= subT[x].query(k - d);
        }
        return res;
    }
    int main() {
        int n = read();
        int m = read();
        for(int i = 1; i <= n; ++i)
            w[i] = read();
        for(int i = 1; i < n; ++i) {
            int u = read();
            int v = read();
            addEdge(u, v);
            addEdge(v, u);
        }
        buildTree(1);
        top[1] = 1;
        buildChain(1);
        divide(getRoot(1, n));
        int lastAns = 0;
        for(int t = 1; t <= m; ++t) {
            int op = read();
            int x = read() ^ lastAns;
            int y = read() ^ lastAns;
            if(op) {
                int delta = y - w[x];
                w[x] = y;
                for(int i = 0; i < uq[x].size(); ++i) {
                    std::pair<BIT*, int> p = uq[x][i];
                    p.first->modify(p.second, delta);
                }
            } else {
                src = x, k = y;
                printf("%d\n", lastAns = query(x, y));
            }
```

```
    }
    return 0;
}
```

11.7.2.4 自底向上统计的点分治 Cache

例题:「SDOI2016」模式字符串

此题我使用了自底向上的 DFS 统计方法, 从模式串左右边界开始匹配。~~事实上也有自顶向下的 Hash 统计方法~~。交上去后发现运行时间是时间的 1.5 倍, 瓶颈在于 DFS 匹配时的 std::vector 的频繁操作。

注意到自底向上统计有一个自顶向下统计所没有的优势: 不变的子树的 DFS 结果不变。那么可以考虑 cache 计算结果。但是由于点分治过程可能会切断子树的某一部分而导致子树变化。注意到 DFS 的子树大小是不断减小的, 可以在 solve 过程中预先处理以分治点为根的子树中每个点的子树大小, 同时记录 cache 的子树大小, 根据这两个数据判断是否要刷新 cache。

Chapter 12

最短路相关问题

12.1 单/多源最短路 377
12.1.1 SPFA 377
12.1.2 算法优化 378
12.1.3 Dijkstra 378
12.1.4 01 最短路 378
12.1.5 Johnson 算法 381
12.2 差分约束系统 382
12.2.1 与最短/长路的关系 382
12.2.2 判断是否有可行解 382
12.2.3 求解 385
12.3 k 短路 385
12.3.1 A* 算法 385
12.3.2 可持久化左偏树法 385

12.1 单/多源最短路

12.1.1 SPFA

最坏时间复杂度 $O(VE)$。

严重警告 是 SPFA 让我在 NOI2018Day1 中滚粗的。

SPFA 优化的思路基本上是使队列尽可能接近优先队列。下列优化中 "当前节点" 指被松弛后将入队的节点。

12.1.1.1 SLF 优化

插入队列时, 若当前节点比队首距离更短则插入队首, 否则插入队尾。

12.1.1.2 LLL 优化

若队首超过队列平均距离则将其塞入队尾, 否则进行松弛。

12.1.1.3 堆优化

沿用 LLL 优化的思路：使用距离尽可能小的节点来松弛。使用堆来维护。

12.1.1.4 SLF 带容错

令边权和为 W，若当前节点比队首距离多 $\sqrt{W}$ 才插入队尾。在边权和小的图上表现不错。

12.1.1.5 mcfx 优化

在第 $[2, \sqrt{V}]$ 次访问当前节点时插入队首，否则插入队尾。在网格图上表现优秀，搭配 SLF 带容错优化效果更佳。

12.1.1.6 SLF+Swap

插入队尾后若队首大于队尾则交换首尾。
更多优化与 Hack 参见https://www.zhihu.com/question/292283275。

12.1.2 算法优化

若用节点编号表示状态来求最短路，要具体分析题目，阻止无效状态入队。

12.1.3 Dijkstra

使用二叉堆的最坏时间复杂度 $O((V+E)\lg V)$，在稀疏图中表现良好。稠密图可以使用 $O(V^2)$ 暴力。

强烈安利 大型考试还是使用 Dijkstra 算法好了。
注意 Dijkstra 中不方便的堆内修改可以改为加入新点（标号 + 松弛距离）的形式，出堆时与数组中记录的距离值比较，匹配才进行松弛。当然由于每个节点只会松弛一次，使用 flag 记录是否松弛过也可以正确运行，这个方法在扩展 Dijkstra 中比较好用。

12.1.4 01 最短路

无权图最短路使用 BFS 解决，增加了权为 0 的边后，仅需稍微修改入队规则：维护双端队列，若边权为 0 则加入队首，否则加入队尾。时间复杂度 $O(V+E)$。
参考代码（[PA 2011]Journeys）：

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>
namespace IO {
    const int size = 1 << 22;
    char in[size], *IS = in;
    void init() {
        setvbuf(stdin, 0, _IONBF, 0);
        setvbuf(stdout, 0, _IONBF, 0);
        fread(in, 1, size, stdin);
    }
    char getc() {
```

```
        return *IS++;
    }
    char out[size], *OS = out;
    void putc(char ch) {
        *OS++ = ch;
    }
    void uninit() {
        fwrite(out, OS - out, 1, stdout);
    }
}
int read() {
    int res = 0, c;
    do
        c = IO::getc();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = IO::getc();
    }
    return res;
}
void write(int val) {
    if(val >= 10)
        write(val / 10);
    IO::putc('0' + val % 10);
}
const int size = 500005, maxS = size << 2,
          maxv = maxS * 2;
struct Edge {
    int to, nxt;
} E[size * 85];
int last[maxv], cnt = 0;
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v << 1 | w, E[cnt].nxt = last[u];
    last[u] = cnt;
}
#define ls l, m, id << 1
#define rs m + 1, r, id << 1 | 1
int lid[size], iid[maxS], oid[maxS], pcnt = 0;
void build(int l, int r, int id) {
    if(l == r)
        lid[l] = iid[id] = oid[id] = ++pcnt;
    else {
        iid[id] = ++pcnt;
        oid[id] = ++pcnt;
        int m = (l + r) >> 1;
        build(ls);
        build(rs);
        addEdge(iid[id], iid[id << 1], 0);
```

```
            addEdge(iid[id], iid[id << 1 | 1], 0);
            addEdge(oid[id << 1], oid[id], 0);
            addEdge(oid[id << 1 | 1], oid[id], 0);
        }
    }
    std::pair<int, int> A[2][100], *ptr;
    void scan(int l, int r, int id, int nl, int nr) {
        if(nl <= l && r <= nr)
            *(ptr++) = std::make_pair(iid[id], oid[id]);
        else {
            int m = (l + r) >> 1;
            if(nl <= m)
                scan(ls, nl, nr);
            if(m < nr)
                scan(rs, nl, nr);
        }
    }
    int dis[maxv], q[maxv];
    bool flag[maxv];
    void SSSP(int s) {
        memset(dis + 1, 0x3f, sizeof(int) * pcnt);
        int b = 0, e = 1;
        q[0] = s, dis[s] = 0;
        while(b != e) {
            int u = q[b++];
            if(b == maxv)
                b = 0;
            if(flag[u])
                continue;
            flag[u] = true;
            for(int i = last[u]; i; i = E[i].nxt) {
                int v = E[i].to >> 1, w = E[i].to & 1;
                if(dis[v] > dis[u] + w) {
                    dis[v] = dis[u] + w;
                    if(w) {
                        q[e++] = v;
                        if(e == maxv)
                            e = 0;
                    } else {
                        if(--b == -1)
                            b = maxv - 1;
                        q[b] = v;
                    }
                }
            }
        }
    }
    int main() {
        IO::init();
        int n = read();
```

```
    int m = read();
    int p = read();
    build(1, n, 1);
    for(int i = 1; i <= m; ++i) {
        int a = read();
        int b = read();
        ptr = A[0];
        scan(1, n, 1, a, b);
        int sizA = ptr - A[0];
        int c = read();
        int d = read();
        ptr = A[1];
        scan(1, n, 1, c, d);
        int sizB = ptr - A[1];
        for(int j = 0; j < sizA; ++j) {
            int iu = A[0][j].first,
                ou = A[0][j].second;
            for(int k = 0; k < sizB; ++k) {
                int iv = A[1][k].first;
                int ov = A[1][k].second;
                addEdge(ou, iv, 1);
                addEdge(ov, iu, 1);
            }
        }
    }
    SSSP(lid[p]);
    for(int i = 1; i <= n; ++i) {
        write(dis[lid[i]]);
        IO::putc('\n');
    }
    IO::uninit();
    return 0;
}
```

注意队列中可能存在重复元素，循环队列要开到与边集一样大。

12.1.5 Johnson 算法

Johnson 算法用于求所有节点对间最短路径。其主要思想是将原图转换为无负权边的图。

算法步骤如下：

1. 新建节点 s，将 s 连向原图所有点，边权为 0。
2. 跑 SPFA 计算以 s 为源点的最短路，同时检查负权环。记点 u 的距离标号为 $h[u]$，满足三角不等式 $h[u]+w[u][v]\geq h[v]$，变形得 $h[u]+w[u][v]-h[v]\geq 0$。令新边权为 $h[u]+w[u][v]-h[v]$，满足 Dijkstra 需要无负权边的条件。
3. 以每个节点为源点跑 Dijkstra 填充最短距离矩阵。记以点 u 为源点时，点 u 到点 v 的实际距离为 $md[v]$，Dijkstra 结果为 $dis[v]$，将边权加和后得 $dis[v]=md[v]+h[u]-h[v]$，变形得 $md[v]=dis[v]-h[u]+h[v]$。

第 2.3.1节所述方法就是该算法的应用之一。该算法在稀疏图上表现比 Floyd 算法好，时间复杂度 $O(VE\lg V)$。

12.2 差分约束系统

差分约束系统是一堆形如 $x_i - x_j \le c_k$ 的不等式组成的不等式组。
$x_i - x_j = c_k$ 等价于 $x_i - x_j \le c_k, x_j - x_i \le -c_k$。

12.2.1 与最短/长路的关系

对不等式 $x_i - x_j \le c_k$ 进行变形得 $x_i \le x_j + c_k$，把 x_i 看做点 i 到源点的距离，该不等式就是三角不等式，等价于从点 j 向点 i 连了一条距离为 c_k 边。如此便可以将差分约束问题转换为最短路问题。最长路的分析类似。

12.2.2 判断是否有可行解

定理 12.1 当图中存在负权环路时，该差分约束系统无可行解。

证明：从负权环路中拆出一条边，记边为 $(u \to\!> v\ w_1)$，其它边合并得 $(v \to u, w_2)$，有 $w_1 + w_2 < 0$，将边还原成不等式组：

$$\begin{aligned} x_v - x_u &\le w_1 \\ x_u - x_v &\le w_2 \end{aligned}$$

两式相加得 $0 \le w_1 + w_2$，与 $w_1 + w_2 < 0$ 矛盾。

使用 SPFA 判负环有 DFS 与 BFS 两种方法。

首先用一个超级源连接所有点，保证图的连通，边权具体情况具体分析。

- DFS-SPFA: 松弛时递归下去，当存在环时说明存在负权环。~~使用 setjmp/longjmp 较为方便~~。
- BFS-SPFA: 当某节点**最短路点数**大于等于节点总数时，存在负权环。使用最短路点数而不是入队次数判断可以避免某些 SPFA 优化影响算法正确性。

一般使用 DFS 法(跑得飞快)，根据姜碧野的论文《SPFA 算法的优化与应用》，还可以使用贪心预处理(松弛 1 次就退出)与迭代加深法加速负环查找。注意将初始距离设为 0 并不影响算法正确性，但要以每个点为起点 DFS(负环上必有一条可以松弛的负权边)。

代码如下：

```
0 #include <csetjmp>
  #include <cstdio>
  #include <cstring>
  int read() {
      int res = 0, c;
      bool flag = false;
      do {
          c = getchar();
          flag |= c == '-';
      } while(c < '0' || c > '9');
10    while('0' <= c && c <= '9') {
```

```
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
const int maxn = 2005, maxm = 6005;
struct Edge {
    int to, nxt, w;
} E[maxm];
int cnt, last[maxn];
void addEdge(int u, int v, int w) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u], E[cnt].w = w;
    last[u] = cnt;
}
int dis[maxn], n;
bool flag[maxn];
jmp_buf buf;
bool pre(int u) {
    int cd = dis[u];
    flag[u] = true;
    bool res = false;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to, nd = cd + E[i].w;
        if(dis[v] > nd) {
            if(flag[v])
                longjmp(buf, 1);
            dis[v] = nd;
            pre(v);
            res = true;
            break;
        }
    }
    flag[u] = false;
    return res;
}
void DFS(int u, int d) {
    if(d) {
        flag[u] = true;
        int cd = dis[u];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to, nd = cd + E[i].w;
            if(dis[v] > nd) {
                if(flag[v])
                    longjmp(buf, 1);
                dis[v] = nd;
                DFS(v, d - 1);
            }
        }
        flag[u] = false;
```

```
    }
}
bool vis[maxn];
void color(int u) {
    vis[u] = true;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(!vis[v])
            color(v);
    }
}
bool foo() {
    n = read();
    int m = read();
    memset(last, 0, sizeof(int) * (n + 1));
    memset(dis, 0, sizeof(int) * (n + 1));
    memset(flag, 0, sizeof(bool) * (n + 1));
    memset(vis, 0, sizeof(bool) * (n + 1));
    cnt = 0;
    while(m--) {
        int u = read();
        int v = read();
        int w = read();
        addEdge(u, v, w);
        if(w >= 0)
            addEdge(v, u, w);
    }
    color(1);
    switch(setjmp(buf)) {
        case 0: {
            for(int i = 1; i <= n; ++i)
                if(vis[i])
                    while(pre(i))
                        ;
            for(int k = 1; k <= n * 2; k <<= 1)
                for(int i = 1; i <= n; ++i)
                    if(vis[i])
                        DFS(i, k);
        } break;
        case 1:
            return true;
    }
    return false;
}
int main() {
    int t = read();
    while(t--)
        puts(foo() ? "YE5" : "N0");
    return 0;
}
```

12.2.2.1 例题

Luogu P4578 [FJOI2018] 所罗门王的宝藏 [1]

看上去可以使用高斯消元法求解系数矩阵为稀疏矩阵的线性方程组。进一步分析发现，若将 y 方向操作的左右取反，可以得到 $x_i - y_j = c_k$ 的形式。将该形式拆成最短路的差分约束形式：

$$\begin{aligned} x_i - y_j &\leq c_k \\ y_j - x_i &\leq -c_k \end{aligned}$$

双向连边后 SPFA 判负环。

12.2.3 求解

最短路的距离就是一组可行解，对这组可行解任意加减一个常数可以得到另一个可行解。一般要求最小非负解。

12.3 k 短路

12.3.1 A* 算法

主要思想是使用启发式函数来修正关键字，以达到加速效果。

步骤如下：

1. 将启发式函数设为当前节点到目标节点的最短距离，SSSP 预处理。
2. 维护一个优先队列（关键字为估价函数 = 当前距离 + 到目标点的最短路），加入起点。
3. 选取估价函数最小的点松弛（不比较距离直接加入堆中）。根据以下定理：

   定理 12.2 一个点第 k 次出队后，当前距离就是它到起点的第 k 短路。

   在目标节点出队时计算是否为第 k 次出队，满足就返回。
4. 若中途不返回则说明不存在第 k 短路，返回无解。

注意判断 $s = t$ 和 $s - t$ 不连通的情况。

以上内容参考了 Z_Mendez 的博客[2]。

12.3.2 可持久化左偏树法

步骤如下：

1. 对反图做 SSSP；
2. 在反图上建出最短路树，使用 DFS 递归预处理每个节点在树边路径上的所有置换为非树边的方案（使用小根堆维护）：

[1]【P4578】[FJOI2018] 所罗门王的宝藏 - 洛谷 https://www.luogu.org/problemnew/show/P4578

[2]A* 算法—第 K 短路 - This is Mendez. https://blog.csdn.net/z_mendez/article/details/47057461

(a) 如果当前点不是终点

i. 选取非树边(注意有多条边可被当做树边的情况,此时只能选一条边当做树边,而且仅 $v == fa[u]$ 不能判断该情况,因为可能有重边),计算将当前边置换成非树边后再走最短路增加的距离(注意不可到达终点的情况),加入该节点的堆;

ii. 将父节点的堆合并到自己的堆,因为自己的方案也包括父亲的方案。

注意:

- 合并时可以使用克隆开关或者先把自己的堆 id 初始化为父节点的堆。
- 插入堆中的还有转移点的编号。

(b) 选取树边递归预处理(注意有多条树边的情况,此时要选择所有的点未处理的树边)。

3. 接下来计算 k 短路:

(a) 首先考虑最短路;

(b) 使用优先队列维护小根堆(堆的节点编号,距离),把次短路的信息加入优先队列;

(c) 第 k 次从优先队列中弹出的就是第 k 短路,然后往优先队列中加入更长的路径:

- 从转移点开始不走最短路:使用转移点的堆转移,加入该堆的根作为候选方案。
- 更换转移点:使用原堆继续转移,即加入左右儿子。

这种方法保证了优先队列的小规模,而且更新到优先队列中的方案都比原方案小保证了正确性。

时间复杂度 $O((V+E)\lg V + E\lg E + k\lg k)$。
模板(Luogu P2483 [SDOI2010] 魔法猪学院[3]):

```
// P2483
#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <queue>
typedef double FT;
const FT inf = 1e20, eps = 1e-8;
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
```

[3]【P2483】【模板】k 短路([SDOI2010] 魔法猪学院)- 洛谷 https://www.luogu.org/problemnew/show/P2483

```
FT readFT() {
    int c;
    do
        c = getchar();
    while(!isgraph(c));
    char str[32];
    int pos = 0;
    while(isgraph(c)) {
        str[pos++] = c;
        c = getchar();
    }
    str[pos] = '\0';
    return strtod(str, 0);
}
bool equalZero(FT x) {
    return fabs(x) < eps;
}
const int size = 5005;
struct G {
    struct Edge {
        int to, nxt;
        FT w;
    } E[200005];
    int last[size], cnt;
    G() : cnt(0) {}
    void addEdge(int u, int v, FT w) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u],
        E[cnt].w = w;
        last[u] = cnt;
    }
} g1, g2;
bool flag[size];
int q[size];
FT dis[size];
void SPFA(int n) {
    for(int i = 1; i < n; ++i)
        dis[i] = inf;
    dis[n] = 0.0, q[0] = n, flag[n] = true;
    int b = 0, e = 1;
    while(b != e) {
        int u = q[b++];
        if(b == size)
            b = 0;
        flag[u] = false;
        for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
            int v = g1.E[i].to;
            FT cd = dis[u] + g1.E[i].w;
            if(cd < dis[v]) {
                dis[v] = cd;
```

```
                if(!flag[v]) {
                    flag[v] = true;
                    q[e++] = v;
                    if(e == size)
                        e = 0;
                }
            }
        }
    }
}
struct Node {
    int ls, rs, dis, p;
    FT val;
} T[size * 900];
int root[size], icnt = 0;
bool enableClone = true;
int cloneNode(int src) {
    if(enableClone) {
        int id = ++icnt;
        T[id] = T[src];
        return id;
    }
    return src;
}
int merge(int u, int v) {
    if(u && v) {
        if(T[u].val > T[v].val)
            std::swap(u, v);
        int w = cloneNode(u);
        T[w].rs = merge(T[w].rs, v);
        if(T[T[w].ls].dis < T[T[w].rs].dis)
            std::swap(T[w].ls, T[w].rs);
        T[w].dis = T[T[w].rs].dis + 1;
        return w;
    }
    return u | v;
}
int insert(int src, FT val, int p) {
    int id = ++icnt;
    T[id].val = val;
    T[id].p = p;
    return merge(src, id);
}
int n, fa[size];
void DFS(int u) {
    if(u != n) {
        bool haveEdge = false;
        enableClone = false;
        for(int i = g2.last[u]; i; i = g2.E[i].nxt) {
            int v = g2.E[i].to;
```

```
            FT delta = dis[v] + g2.E[i].w - dis[u];
            if(v == fa[u] && equalZero(delta) &&
               !haveEdge)
                haveEdge = true;
            else
                root[u] = insert(root[u], delta, v);
        }
        enableClone = true;
        root[u] = merge(root[u], root[fa[u]]);
    }
    for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
        int v = g1.E[i].to;
        FT delta = dis[u] + g1.E[i].w - dis[v];
        if(!fa[v] && equalZero(delta)) {
            fa[v] = u;
            DFS(v);
        }
    }
}
struct Info {
    FT dis;
    int id;
    Info(int id, FT dis) : dis(dis), id(id) {}
    bool operator<(const Info& rhs) const {
        return dis > rhs.dis;
    }
};
int main() {
    n = read();
    int m = read();
    FT e = readFT();
    while(m--) {
        int u = read();
        int v = read();
        FT w = readFT();
        g1.addEdge(v, u, w);
        g2.addEdge(u, v, w);
    }
    SPFA(n);
    fa[n] = n;
    DFS(n);
    int res = 0;
    if(e >= dis[1]) {
        e -= dis[1], ++res;
        std::priority_queue<Info> heap;
        heap.push(
            Info(root[1], dis[1] + T[root[1]].val));
        while(heap.size() && e >= heap.top().dis) {
            int u = heap.top().id;
            int v = root[T[u].p];
```

```
170             FT d = heap.top().dis;
                heap.pop();
                e -= d;
                ++res;
                if(v)
                    heap.push(Info(v, d + T[v].val));
                FT base = d - T[u].val;
                if(T[u].ls)
                    heap.push(Info(T[u].ls,
                                   base + T[T[u].ls].val));
180             if(T[u].rs)
                    heap.push(Info(T[u].rs,
                                   base + T[T[u].rs].val));
            }
        }
        printf("%d\n", res);
        return 0;
    }
```

以上内容参考了 litble[4]的博客。

[4]HDU5960 可持久化左偏树 k 短路问题 - litble 的成 (tui) 长 (fei) 史 https://blog.csdn.net/litble/article/details/79171311

Chapter 13

生成树

13.1 最小生成树 391
13.1.1 Kruskal 与 LCT 391
13.1.2 动态 MST 392
13.1.3 Prim 算法 396
13.1.4 次小生成树 397
13.1.5 生成树性质 397
13.1.6 Boruvka 算法 399
13.1.7 单点度限制最小生成树 402
13.2 Kruskal 重构树 402
13.2.1 构造 402
13.2.2 应用 402
13.3 曼哈顿距离 MST 403
13.3.1 分析 403
13.3.2 实现 403
13.4 Matrix-Tree 定理 405
13.4.1 基本定义 405
13.4.2 扩展 405
13.5 斯坦纳树 407

13.1 最小生成树

13.1.1 Kruskal 与 LCT

Luogu P4234 最小差值生成树[1]

按照 Kruskal 的处理顺序处理生成树, 当遇到环时删掉环上权值最小的边, 然后连上自己。

[1]【P4234】最小差值生成树 - 洛谷 https://www.luogu.org/problemnew/show/P4234

13.1.2 动态 MST

支持插入边, 删除边, 修改边权, 每次修改后回答 MST 的代价, 可离线。

首先处理涉及到的所有边, 边不存在时记其边权为 ∞, 那么插入边和删除边都可以转化为修改边权解决。对应的题目为 [HNOI2010] 城市建设。

若边权是不增的, 很容易使用 LCT 维护树形结构, 对于非树边, 每次查询两端在 MST 上的链上最大权边, 讨论是否将其换下。

对于边权可增的情况, 若增权的边在 MST 上, 则需要查询跨连通块的最小非树边, 并讨论是否替换。这个询问不好做。注意到修改可离线, 考虑使用分治解决。

在这 k 个修改操作中, 产生了 k 个 MST, 有些边一定不出现在 MST 中, 有些边一定出现在 MST 中。一定不出现的边可以直接删除, 一定出现的边可以使用并查集连起来, 分别减少了边数和点数, 缩减图的规模。

处理无用边: 将有修改的边标记为 ∞, 做 MST, 删去无修改且不在 MST 中的边。

处理必须边: 将有修改的边标记为 $-\infty$, 做 MST, 连接无修改且在 MST 中的边。

每次处理 $[l, r]$ 区间内的修改, 然后递归到左右部分处理, 每次递归时都能缩减图的规模。当缩减到只有一个修改时暴力对剩下的边做 MST。递归右边前要执行左边的修改。

参考代码:

```
#include <algorithm>
#include <cstdio>
#include <vector>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 50005, inf = 1 << 30,
          magic = -(0x35fa35c);
typedef long long Int64;
struct Edge {
    int id, u, v, w;
    bool operator<(const Edge& rhs) const {
        return w < rhs.w;
    }
};
bool cmpId(const Edge& a, const Edge& b) {
    return a.id < b.id;
}
int find(std::vector<int>& fa, int x) {
    return fa[x] ? fa[x] = find(fa, fa[x]) : x;
}
void merge(std::vector<int>& fa, std::vector<int>& rk,
           int u, int v) {
    if(rk[u] < rk[v])
        fa[u] = v;
```

```
        else {
            fa[v] = u;
            if(rk[u] == rk[v])
                ++rk[u];
        }
    }
    struct Info {
        std::vector<Edge> E;
        Int64 sum;
        int vsiz;
    };
    void MST(Info& info) {
        std::vector<Edge>& E = info.E;
        std::sort(E.begin(), E.end());
        std::vector<int> fa(info.vsiz + 5),
            rk(info.vsiz + 5);
        int cnt = info.vsiz - 1;
        for(int i = 0; i < E.size(); ++i) {
            int u = find(fa, E[i].u), v = find(fa, E[i].v);
            if(u != v) {
                info.sum += E[i].w;
                E[i].w = magic;
                merge(fa, rk, u, v);
                if(--cnt == 0)
                    break;
            }
        }
    }
    struct Op {
        int k, w;
    } Op[size];
    Int64 ans[size];
    void solve(int l, int r, Info& info) {
        if(l == r) {
            for(int i = 0; i < info.E.size(); ++i)
                if(info.E[i].id == Op[l].k)
                    info.E[i].w = Op[l].w;
            MST(info);
            ans[l] = info.sum;
        } else {
            std::vector<int> eid;
            eid.reserve(r - l + 1);
            for(int i = l; i <= r; ++i)
                eid.push_back(Op[i].k);
            std::sort(eid.begin(), eid.end());
            eid.erase(std::unique(eid.begin(), eid.end()),
                      eid.end());
            // R
            {
                Info tmp;
```

```
        for(int i = 0; i < info.E.size(); ++i) {
            if(!std::binary_search(eid.begin(),
                                   eid.end(),
                                   info.E[i].id))
                tmp.E.push_back(info.E[i]);
        }
        tmp.vsiz = info.vsiz;
        MST(tmp);
        std::vector<int> ueid;
        for(int i = 0; i < tmp.E.size(); ++i)
            if(tmp.E[i].w == magic)
                ueid.push_back(tmp.E[i].id);
        std::sort(ueid.begin(), ueid.end());
        int cur = 0;
        for(int i = 0; i < info.E.size(); ++i) {
            int id = info.E[i].id;
            if(std::binary_search(eid.begin(),
                                  eid.end(), id) ||
               std::binary_search(ueid.begin(),
                                  ueid.end(), id))
                info.E[cur++] = info.E[i];
        }
        info.E.resize(cur);
    }
    // C
    {
        Info tmp = info;
        for(int i = 0; i < tmp.E.size(); ++i) {
            if(std::binary_search(eid.begin(),
                                  eid.end(),
                                  tmp.E[i].id))
                tmp.E[i].w = -1;
        }
        MST(tmp);
        std::vector<int> ueid;
        for(int i = 0; i < tmp.E.size(); ++i)
            if(!std::binary_search(eid.begin(),
                                   eid.end(),
                                   tmp.E[i].id) &&
               tmp.E[i].w == magic)
                ueid.push_back(tmp.E[i].id);
        std::sort(ueid.begin(), ueid.end());
        std::vector<int> fa(info.vsiz + 5),
            rk(info.vsiz + 5);
        int cur = 0;
        for(int i = 0; i < info.E.size(); ++i) {
            int id = info.E[i].id;
            if(std::binary_search(
                   ueid.begin(), ueid.end(), id)) {
                info.sum += info.E[i].w;
```

```
                    merge(fa, rk,
                          find(fa, info.E[i].u),
                          find(fa, info.E[i].v));
                } else
                    info.E[cur++] = info.E[i];
            }
            info.E.resize(cur);
            std::vector<int> pid;
            for(int i = 1; i <= info.vsiz; ++i)
                if(find(fa, i) == i)
                    pid.push_back(i);
            info.vsiz = pid.size();
            for(int i = 0; i < info.E.size(); ++i) {
                int u = find(fa, info.E[i].u);
                int v = find(fa, info.E[i].v);
                info.E[i].u =
                    std::lower_bound(pid.begin(),
                                     pid.end(), u) -
                    pid.begin() + 1;
                info.E[i].v =
                    std::lower_bound(pid.begin(),
                                     pid.end(), v) -
                    pid.begin() + 1;
            }
        }
        int m = (l + r) >> 1;
        Info tmp = info;
        solve(l, m, tmp);
        std::sort(info.E.begin(), info.E.end(), cmpId);
        for(int i = l; i <= m; ++i) {
            Edge e;
            e.id = Op[i].k;
            std::vector<Edge>::iterator p =
                std::lower_bound(info.E.begin(),
                                 info.E.end(), e,
                                 cmpId);
            if(p != info.E.end() && p->id == e.id)
                p->w = Op[i].w;
        }
        solve(m + 1, r, info);
    }
}
int main() {
    int n = read();
    int m = read();
    int q = read();
    Info info;
    info.vsiz = n;
    info.sum = 0;
    info.E.resize(m);
```

```
    for(int i = 0; i < m; ++i) {
        info.E[i].id = i + 1;
        info.E[i].u = read();
        info.E[i].v = read();
        info.E[i].w = read();
    }
    for(int i = 1; i <= q; ++i) {
        Op[i].k = read();
        Op[i].w = read();
    }
    solve(1, q, info);
    for(int i = 1; i <= q; ++i)
        printf("%lld\n", ans[i]);
    return 0;
}
```

上述内容参考了顾昱洲的课件《浅谈一类分治算法》。

13.1.3 Prim 算法

从一个点开始贪心地连接最短的可扩展边, 直至每个点都被连接。
代码:

```
struct Info{
    int u,d;
    Info(int u,int d):u(u),d(d){}
    bool operator<(const Info& rhs) const {
        return d>rhs.d;
    }
};
bool flag[size];
int prim(int n) {
    flag[1]=true;
    std::priority_queue<Info> heap;
    for(int i=last[1];i;i=E[i].nxt)
        heap.push(Info(E[i].to,E[i].w));
    int cnt=1,sum=0;
    while(cnt<n && heap.size()) {
        int u=heap.top().u;
        int d=heap.top().d;
        heap.pop();
        if(!flag[u]) {
            flag[u]=true;
            sum+=E[i].w;
            for(int i=last[u];i;i=E[i].nxt) {
                int v=E[i].to;
                if(!flag[v])
                    heap.push(Info(v,E[i].w));
            }
        }
    }
```

```
    return cnt==n?sum:-1;
}
```

当边数较大时, 可以考虑使用 Prim 或优先队列 +Kruskal(参见第 19.1.12 节)。

13.1.4 次小生成树

步骤如下:

1. 构造最小生成树, 标记并连接树边;
2. 对生成树进行树链剖分, 构造线段树, 使其可以 $O(\lg n)$ 查询到链上最大权;
3. 枚举非树边, 计算其替代链上的最大边后的代价, 更新答案。

对于严格次小生成树, 需要维护链上最大权与严格次大权。

13.1.5 生成树性质

以下性质用于解决最小生成树计数问题:

性质 13.1 最小生成树中, 不同权值边的数量固定。

性质 13.2 *Kruskal* 算法中, 处理完某一权值的边后, 连通块相同。

13.1.5.1 例题

Luogu P4208 [JSOI2008] 最小生成树计数[2]
根据以上两个性质可以把每种权值的边分别计算, 然后使用乘法原理组合即为答案。
步骤如下:

1. 首先使用常规 Kruskal 计算每种权值边的数量 c_i 以及等权边的分布区间;
2. 对于每一种权值, 计算用完 c_i 条边的方案数。由于题中每种权值的边数不超过 10, 可以枚举每条边选还是不选, 在回溯时直接令连接的两点的根的父亲重设为自己, 时间复杂度近似为 $O(2^{c_i})$。模拟对该权值边做 Kruskal 的过程。
3. 把每种权值的方案数乘起来就是方案数。

代码如下:

```
0 #include <algorithm>
  #include <cstdio>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
      while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
          c = getchar();
10    }
```

[2]【P4208】[JSOI2008] 最小生成树计数 - 洛谷 https://www.luogu.org/problemnew/show/P4208

```
    return res;
}
const int size = 105, maxm = 1005, mod = 31011;
struct Edge {
    int u, v, w;
    bool operator<(const Edge& rhs) const {
        return w < rhs.w;
    }
} E[maxm];
int fa[size], end[maxm], refCount[maxm], cnt;
int find(int u) {
    return fa[u] == u ? u : find(fa[u]);
}
void DFS(int p, int e, int rem) {
    if(rem) {
        if(p < e) {
            int u = find(E[p].u), v = find(E[p].v);
            if(u != v) {
                fa[u] = v;
                DFS(p + 1, e, rem - 1);
                fa[u] = u, fa[v] = v;
            }
            DFS(p + 1, e, rem);
        }
    } else
        ++cnt;
}
int main() {
    int n = read();
    int m = read();
    for(int i = 1; i <= n; ++i)
        fa[i] = i;
    for(int i = 1; i <= m; ++i) {
        E[i].u = read();
        E[i].v = read();
        E[i].w = read();
    }
    std::sort(E + 1, E + m + 1);
    int ecnt = 0, wcnt = 0;
    end[0] = 1;
    for(int i = 1; i <= m; ++i) {
        if(E[i].w != E[i - 1].w)
            end[wcnt++] = i;
        int u = find(E[i].u), v = find(E[i].v);
        if(u != v) {
            fa[u] = v;
            ++ecnt;
            ++refCount[wcnt];
        }
    }
```

```
    end[wcnt] = m + 1;
    if(ecnt == n - 1) {
        long long ans = 1;
        for(int i = 1; i <= n; ++i)
            fa[i] = i;
        int cur = 1;
        for(int i = 1; i <= wcnt; ++i) {
            cnt = 0;
            DFS(end[i - 1], end[i], refCount[i]);
            ans = ans * cnt % mod;
            while(cur < end[i]) {
                int u = find(E[cur].u),
                    v = find(E[cur].v);
                if(u != v)
                    fa[u] = v;
                ++cur;
            }
        }
        printf("%lld\n", ans);
    } else
        puts("0");
    return 0;
}
```

另一种方法：每次计算小连通块-> 大连通块的过程，对于每个大连通块单独使用 Matrix-Tree 定理求方案数，方案数之积即为答案。

最小生成树计数问题参考了 clover_hxy 的博客[3]。

13.1.6 Boruvka 算法

当边数较大时，还可以使用 Boruvka 算法，该算法像多路的 Kruskal 算法。

算法步骤如下：

1. 把每个点初始化为一个连通块。
2. 对每个连通块查询其连出的最小权值边。
3. 加入所有边，若边权两两不同则不可能出现环。一般用并查集判环。
4. 重复步骤 2 直至生成树构造完毕。

由于每次迭代后连通块数至少减少一半，共需合并 $O(\lg n)$ 次。每次合并后由于连通块的变化而重建数据结构的复杂度是可以接受的。插入时需要把连通块编号存入，维护连通块编号的最值，以便于查询时排除块内边。

模板（CF888G Xor-MST）：

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
```

[3]bzoj 1016: [JSOI2008] 最小生成树计数（矩阵树定理 + 最小生成树）https://blog.csdn.net/clover_hxy/article/details/69397184

```
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 200005;
struct Node {
    int mini, maxi, c[2];
    void update(int id) {
        if(id < mini)
            mini = id;
        else if(id > maxi)
            maxi = id;
    }
} T[size * 30];
int tsiz;
int alloc(int pid) {
    int id = ++tsiz;
    T[id].maxi = T[id].mini = pid;
    T[id].c[0] = T[id].c[1] = 0;
    return id;
}
void insert(int p, int id, int val) {
    T[p].update(id);
    for(int i = 29; i >= 0; --i) {
        int& c = T[p].c[(val >> i) & 1];
        if(!c)
            c = alloc(id);
        else
            T[c].update(id);
        p = c;
    }
}
struct Res {
    int v, w;
    Res() {}
    Res(int v, int w) : v(v), w(w) {}
};
Res query(int p, int id, int val) {
    Res res(0, 0);
    for(int i = 29; i >= 0; --i) {
        int mask = (val >> i) & 1;
        int u = T[p].c[mask];
        if(u &&
           !(T[u].maxi == T[u].mini &&
             T[u].maxi == id))
            p = u;
```

```
            else
                p = T[p].c[mask ^ 1], res.w |= 1 << i;
        }
        res.v = (T[p].mini == id ? T[p].maxi : T[p].mini);
        return res;
    }
    int fa[size];
    int find(int x) {
        return fa[x] ? fa[x] = find(fa[x]) : x;
    }
    int A[size];
    int rebuild(int n) {
        tsiz = 0;
        int rt = alloc(0);
        T[rt].maxi = n + 1;
        for(int i = 1; i <= n; ++i)
            insert(rt, find(i), A[i]);
        return rt;
    }
    Res sol[size];
    int main() {
        int n = read();
        for(int i = 1; i <= n; ++i)
            A[i] = read();
        int bcnt = n;
        long long ans = 0;
        while(bcnt > 1) {
            int rt = rebuild(n);
            for(int i = 1; i <= n; ++i)
                sol[i] = Res(0, 1 << 30);
            for(int i = 1; i <= n; ++i) {
                int fi = find(i);
                Res csol = query(rt, fi, A[i]);
                if(sol[fi].w > csol.w)
                    sol[fi] = csol;
            }
            for(int i = 1; i <= n; ++i)
                if(find(i) == i) {
                    int fv = find(sol[i].v);
                    if(fv != i) {
                        fa[i] = fv;
                        ans += sol[i].w;
                        if(--bcnt == 1)
                            break;
                    }
                }
        }
        printf("%lld\n", ans);
        return 0;
    }
```

13.1.7 单点度限制最小生成树

要求指定点 v_0 的度数为 k, 求 MST。

首先把原图去掉 v_0 求最小生成森林, 此时这些连通块只能通过 v_0 与之相连。若连通块数 $> k$, 则无解。然后每个连通块内选择一条边权最小的边连向 v_0。若连通块数 $< k$ 则不断选择块外未选择边连上 v_0, 然后切断对应连通块。

该内容参考了唐文斌在 WC2012 上的讲稿《图论专题之生成树》。

13.2 Kruskal 重构树

13.2.1 构造

在做 Kruskal 时, 将合并两点所在集合的操作改为将两点所在集合连到一个新的点上(代表边), 新点的权为边权, 代表点的叶子节点的权为 0, 这样就可以构造出一棵 Kruskal 重构树, 用来处理边权最值问题。

由构造过程可推出 Kruskal 重构树的性质:

性质 13.3 最小生成树与 *Kruskal* 重构树两点路径上的边权(点权)最大值相等。

性质 13.4 *Kruskal* 重构树中子节点的权小于等于父节点的权。换句话说, *Kruskal* 重构树同时也是一个大根堆。

由上述性质可得到在 Kruskal 重构树中询问边权最大值的方法: 计算两点的 LCA, LCA 的点权就是原生成树上边权最大值。

以上内容参考了 Coco_T_ 的博客[4]。

13.2.2 应用

13.2.2.1 例题

Luogu P4768 [NOI2018] 归程[5]

很容易看出这题求的是汽车可到达节点到原节点的最短路的最小值。

首先使用 Dijkstra 预处理最短路 (**不要用 SPFA!!! 不要用 SPFA!!! 不要用 SPFA!!!**)。

可持久化并查集法 一个很简单的思路是按照海拔高度从大到小连边, 使用可持久化并查集记录连通性, 同时维护连通块内最短路最小值, 预处理后查询集合最值。

Kruskal 重构树法 由性质 13.4不难想到对该图按海拔高度做最大生成树, 这样就保证了 Kruskal 重构树的父节点点权小于等于子节点点权。使用倍增法可以跳到满足要求的的最浅的祖先。这个祖先就代表了满足海拔要求的最大连通集合。对每个点存储子树叶子节点(即生成树中的点)的最短路最小值以支持 $O(\lg n)$ 查询。

13.2.2.2 最小瓶颈路

最小瓶颈路求的是无向图中指定两点的路径的边权最大值的最小值。

无向图中任意两点的最小瓶颈路肯定在最小生成树上, 建出 Kruskal 重构树, 根据性质 13.4, 两点的 LCA 即为边权最大值。

---

[4] bzoj3732 Network(Kruskal 重构树)
https://blog.csdn.net/wu_tongtong/article/details/77601523

[5] 【P4768】[NOI2018] 归程 - 洛谷 https://www.luogu.org/problemnew/show/P4768

13.2.2.3 次小生成树

求解次小生成树的思路是使用非树边替换链上最大边, 按照 13.2.2.2节所述可以简洁地求出链上边权最大值。对于严格次小生成树, 对每个点维护子树严格次小值也是很简单的事情(比线段树不知道高到哪里去了)。

13.3 曼哈顿距离 MST

13.3.1 分析

由于 $E = O(V^2)$, 显然不能使用常规 Kruskal 法。

考虑曼哈顿距离的特性, 以每个点为原点建坐标系, 将坐标系每 $45°$ 分成一块区域, 发现这个点到每个区域最多只会连离它最近的点, 因此只需预处理每个点到八个区域的最近点, 边数缩减为 $O(V)$。根据对称性可只考虑一半的区域, 最多加入 $4V$ 条边。

接下来思考如何查询最近点(仅处理右半区域, 以下左右分别指逆时针和顺时针):

- 首先考虑 y 正半轴向右 $45°$ 的点, 发现若以点 P 为原点, 落在该区域的点 Q 满足:

$$\begin{aligned} x_Q \geq x_P \quad &\Rightarrow \quad x_P \leq x_Q \\ x_Q - x_P \leq y_Q - y_P \quad &\Rightarrow \quad x_Q - y_Q \leq x_P - y_P \end{aligned}$$

  仅一个不等式满足等号就足够了, 因为其它区域可以覆盖到这个边界。

  因为 $|PQ| = (x_Q + y_Q) - (x_P + y_P)$, 所以要查询满足要求且 $x + y$ 最小时对应的点。

  按照 x 从大到小排序, 对 $x - y$ 离散, 使用树状数组维护 $x + y$ 前缀最小值与对应的点。

- 对于其它三个区域的点, 可以使用坐标变换得到(连续变换):

  1. y 正半轴向右:直接做;
  2. x 正半轴向左:交换 x, y;
  3. x 正半轴向右:x 取反;
  4. y 负半轴向左:交换 x, y。

以上内容参考了 GGBeng 的博客[6]。

13.3.2 实现

```
0 struct Node {
      int id,val;
      Node(int id,int val):id(id),val(val) {}
      void update(const Node& rhs) {
          if(val>rhs.val)
              *this=rhs;
      }
  } T[size];
```

[6]曼哈顿距离最小生成树 https://www.cnblogs.com/xzxl/p/7237246.html

```
void reset(int siz) {
    for(int i=1;i<=siz;++i)
        T[i].id=-1,T[i].val=1<<30;
}
void modify(int x,int siz,const Node& val) {
    while(x<=siz) {
        T[x].update(val);
        x+=x&-x;
    }
}
Node query(int x) {
    Node val(-1,1<<30);
    while(x) {
        val.update(T[x]);
        x-=x&-x;
    }
    return val;
}
int A[size];
int find(int x,int siz) {
    return std::lower_bound(A+1,A+siz+1,x)-A;
}
struct Pos {
    int x,y,id;
    bool operator<(const Pos& rhs) const {
        return x>rhs.x;
    }
} P[size];
struct Edge {
    int u,v,w;
    Edge(int u,int v,int w):u(u),v(v),w(w) {}
} E[4*size];
int buildGraph(int n) {
    int ecnt=0;
    for(int d=0;d<4;++d) {
        if(d==2) {
            for(int i=1;i<=n;++i)
                P[i].x=-P[i].x;
        }
        else if(d!=0) {
            for(int i=1;i<=n;++i)
                std::swap(P[i].x,P[i].y);
        }
        for(int i=1;i<=n;++i)
            A[i]=P[i].x-P[i].y;
        std::sort(A+1,A+n+1);
        int siz=std::unique(A+1,A+n+1)-(A+1);
        reset(siz);
        std::sort(P+1,P+n+1);
        for(int i=1;i<=n;++i) {
```

```
                int pos=find(P[i].x-P[i].y,siz);
                Node p=query(pos);
60              int sum=P[i].x+P[i].y;
                if(p.id!=-1)
                    E[++ecnt]=Edge(P[i].id,p.id,p.val-sum);
                modify(pos,Node(P[i].id,sum));
            }
        }
        return ecnt;
    }
```

13.4 Matrix-Tree 定理

13.4.1 基本定义

定理 13.5 (Kirchhoff's Matrix Tree Theorem) 一个无向图的生成树个数为度数矩阵(第 u 行第 u 列为点 u 的度数)减邻接矩阵(第 u 行第 v 列为 u,v 之间的边数)去掉第 i 行第 i 列后的行列式值。

根据这个定理, $O(n^3)$ 便可以求解无向图的生成树计数问题。**注意高斯消元时交换两行会使行列式值取反,处理时记录符号或者直接返回其绝对值(仅限于非模意义下求值)。**

13.4.2 扩展

13.4.2.1 完全图生成树

定理 13.6 (Cayley's Formula) 大小为 n 的完全图的生成树个数为 n^{n-2}。

套用 Matrix-Tree 定理或者使用 Purfer 序列可证明。若点与点之间的边数为 m, 方案再乘上 m^{n-1}。

13.4.2.2 有向图生成树计数

邻接矩阵只记录有向边, 度数矩阵只记录入度, 以 s 为根时删去第 s 行第 s 列后求行列式。

13.4.2.3 边权乘积和

把度数矩阵改为与某点相连的边的边权和, 把邻接矩阵的边数改为该边的边权和。若边权为整数则可以将其理解为将权值为 w 的边拆成 w 条边后求生成树数。

在「长乐集训 2017 Day10」生成树求和 加强版 这道题中, 按位拆分后发现需要做不进位加法, 但是矩阵树定理只能做乘法。考虑使用一个三元组表示和为 0,1,2 的种类数, 最后做高斯消元。但是多项式高斯消元并不好做, 可以代入多个点值求高斯消元, 然后插值。注意这里的乘法需要做循环卷积, 可以使用 3 个单位根作为代入点值, 最后计算以 3 为基的 IDFT 插值(暴力计算以点值为多项式系数, 在 3 个单位根的逆上的值, 最后将结果除以 3)。

$x^3=1$ 的求根方法: 首先易得 $x=1$ 是它的根, 那么 x^3-1 可以分解为 $(x-1)A$ 的形式, 做多项式除法可得 $\frac{x^3-1}{x-1}=1+x+x^2$, 解得另外两个根为 $\frac{-1\pm\sqrt{3}i}{2}$。

13.4.2.4 概率扩展

Luogu P3317 [SDOI2014] 重建[7]

图中的每条边都有出现的概率, 求图恰好连成一棵生成树的概率。

答案为每种方案的树边出现的概率和非树边不出现的概率之积的和。将答案除以所有边都不出现的概率, 转化为每种方案的树边出现的概率除以树边不出现的概率的和。由此可以将其转化为边权乘积问题, 即令出现概率为 p 的边的边权为 $\frac{p}{1-p}$, 求完行列式后乘以 $\prod_{i=1}^{m}(1-p)$。注意使用偏移 ε 来防止除零。

代码如下:

```
#include <algorithm>
#include <cmath>
#include <cstdio>
const int size = 60;
double P[size][size];
const double eps = 1e-8;
double clamp(double x) {
    return x > eps ? x : eps;
}
double slove(int n) {
    double res = 1.0;
    for(int i = 1; i <= n; ++i) {
        int base = i;
        for(int j = i + 1; j <= n; ++j)
            if(fabs(P[j][i]) > fabs(P[base][i]))
                base = j;
        if(i != base)
            for(int j = 1; j <= n; ++j)
                std::swap(P[i][j], P[base][j]);
        if(fabs(P[i][i]) < eps)
            return 0.0;
        res *= P[i][i];
        for(int j = i + 1; j <= n; ++j) {
            double t = -P[j][i] / P[i][i];
            for(int k = i; k <= n; ++k)
                P[j][k] += P[i][k] * t;
        }
    }
    return fabs(res);
}
int main() {
    int n;
    scanf("%d", &n);
    double k = 1.0;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= n; ++j) {
            scanf("%lf", &P[i][j]);
```

[7]【P3317】[SDOI2014] 重建 - 洛谷 https://www.luogu.org/problemnew/show/P3317

```
                double x = clamp(1.0 - P[i][j]);
                if(i < j)
                    k *= x;
40              P[i][j] /= x;
            }
        for(int i = 1; i <= n; ++i)
            for(int j = 1; j <= n; ++j)
                if(i != j)
                    P[i][i] -= P[i][j];
        printf("%.8lf\n", slove(n - 1) * k);
        return 0;
    }
```

13.4.2.5 限制边数

图上的边有两种颜色, 限制生成树中一种颜色的边的数量, 求方案数。

令该颜色的边对应 x, 另一种颜色对应 1, 构造多项式, 最后求出的行列式多项式的 x^k 项的系数就对应使用 k 条边的方案数。多项式高斯消元不太方便, 可以先预处理 $|V|$ 个 x 所对应的行列式值, 然后插值出多项式。

上述内容参考了 MoebiusMeow 的博客[8]和 Wikipedia-EN[9]。

13.5 斯坦纳树

最小斯坦纳树求的是将指定的大小为 k 的点集连通的最小代价。

计算步骤如下: 使用状态压缩来描述指定点集的连通状态, 令 $dp[s][i]$ 为以 i 为根, 连通状态为 s 的最小代价。

转移方法有两种:

- 两个不相交集合与同一个点连接, 即

$$dp[s][i] = min\left\{dp[t][i] + dp[s-t][i]\right\}, t \in s$$

- 给集合连入一条新边:

$$dp[s][i] = min\left\{dp[s][j] + w(i,j)\right\}, (i,j) \in E$$

首先令 $dp[1 << j][P[i]] = 0$, 初始化单个点的情况。

可以从小到大枚举连通集合:

1. 枚举子集更新 $dp[i][]$。
2. 将 $dp[][s] \neq \infty$ 的点入队, 使用 SPFA 或 Dijkstra 更新, 注意不要改变连通状态。

枚举子集更新的复杂度为 $n\sum_{i=0}^{k}\binom{k}{i}2^i = n\cdot(1+2)^k = n\cdot 3^k$。

代码如下:

[8]康复计划 #5 Matrix-Tree 定理 (生成树计数) 的另类证明和简单拓展 https://www.cnblogs.com/meowww/p/6485422.html

[9]Kirchhoff's theorem - Wikipedia https://en.wikipedia.org/wiki/Kirchhoff%27s_theorem

```
int dp[1<<maxk][size];
int solve(int n,int k) {
    memset(dp,0x3f,sizeof(dp[0])<<k);
    for(int j=0;j<k;++j)
        for(int i=1;i<=n;++i)
            dp[1<<j][P[i]]=0;
    int end=1<<k;
    for(int s=0;s<end;++s) {
        for(int t=s&(s-1);t;t=s&(t-1))
            for(int i=1;i<=n;++i)
                dp[s][i]=std::min(dp[s][i],
                    dp[t][i]+dp[s^t][i]);
        for(int i=1;i<=n;++i)
            if(dp[s][i]!=inf)
                //push
        SSSP(s);
    }
    int ans=inf;
    for(int i=1;i<=n;++i)
        ans=std::min(ans,dp[end-1][i]);
    return ans;
}
```

13.5.0.1 多组斯坦纳树

给定多个这样的点集, 求最小代价。

重新回想 $dp[s][i]$ 的实际意义而不必考虑 s 中连通节点的组别, $dp[s][i]$ 表示的是我们关注的节点连通状态为 s, 连通块以 i 为根的最小代价。

那么最终结果由一些连通块组成, 把有整组连通的方案当做连通块按照组别状压, 再做一次 DP。

Chapter 14

图论

14.1 割点与桥 410
14.1.1 割点 410
14.1.2 桥 412
14.2 强连通分量 412
14.2.1 定义 412
14.2.2 Tarjan 算法 413
14.3 双连通分量 414
14.3.1 点双连通分量 414
14.3.2 边双连通分量 414
14.4 2-SAT 415
14.4.1 问题描述 415
14.4.2 可行性判定 415
14.4.3 构造方案 415
14.4.4 前后缀优化建图 416
14.5 仙人掌与圆方树 416
14.5.1 仙人掌 416
14.5.2 圆方树 418
14.5.3 广义圆方树 420
14.6 图上路径 421
14.6.1 欧拉路径与欧拉回路 421
14.6.2 哈密尔顿回路 424
14.7 弦图 424
14.7.1 相关概念 424
14.7.2 弦图判定 425
14.7.3 弦图的极大团 428
14.7.4 弦图的点染色 428
14.7.5 弦图的最大独立集与最小团覆盖 430
14.7.6 区间图 430
14.8 带花树 430
14.8.1 无向图最大匹配 430
14.8.2 Micali-Vazirani Algorithm 434
14.8.3 近线性复杂度的随机匹配算法 434

14.8.4 无向图最大权匹配 436
14.9 支配树 436
14.9.1 定义 436
14.9.2 DAG 的支配树 436
14.9.3 一般图的支配树 436
14.10杂讲 436
14.10.1 竞赛图 436
14.10.2 最小平均值环 437
14.10.3 平面图性质 437
14.10.4 拓扑排序判环 437
14.10.5 Lindström–Gessel–Viennot Lemma 438
14.10.6 三元环计数 439

14.1 割点与桥

要注意各类 Tarjan 算法是否需要判断父亲。

14.1.1 割点

若删除**无向连通图**中的一个点及与它相连的边, 使得整个图不连通, 那么称这个点为割点。

查找割点的步骤如下:

1. 从一个点开始 DFS 遍历未被遍历的点;
2. 对于每个点维护其访问时间 dfn 和不经过树边所能访问到的点的访问时间最小值 low;
3. 如果自己不是 DFS 树的根, 若 DFS 树中儿子的 low **不小于**自己的 dfn, 则说明删掉自己后 DFS 树上自己的父亲与自己的儿子不连通, 自己为割点;
4. 如果自己为 DFS 树的根, 并且自己在 DFS 树上有两个及以上的儿子, 说明自己也是割点(若儿子之间连通就会在 DFS 某个出边时将它们都遍历到, 与存在两个及以上儿子矛盾)。

代码如下(求的是每个连通图的割点):

```
0 #include <algorithm>
  #include <cstdio>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
      while('0' <= c && c <= '9') {
          res = res * 10 + c - '0';
          c = getchar();
10    }
      return res;
  }
```

```
const int size = 20005;
struct Edge {
    int to, nxt;
} E[size * 10];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int dfn[size], low[size], ccnt = 0;
bool cut[size];
void DFS(int u, int p) {
    static int icnt = 0;
    dfn[u] = low[u] = ++icnt;
    int child = 0;
    bool flag = false;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p) {
            if(dfn[v])
                low[u] = std::min(low[u], dfn[v]);
            else {
                DFS(v, u);
                ++child;
                low[u] = std::min(low[u], low[v]);
                if(p && dfn[u] <= low[v])
                    flag = true;
            }
        }
    }
    if(!p && child > 1)
        flag = true;
    if(flag) {
        cut[u] = true;
        ++ccnt;
    }
}
int main() {
    int n = read();
    int m = read();
    while(m--) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    for(int i = 1; i <= n; ++i)
        if(!dfn[i])
            DFS(i, 0);
```

```
    printf("%d\n", ccnt);
    for(int i = 1; i <= n; ++i)
        if(cut[i])
            printf("%d ", i);
    return 0;
}
```

14.1.2 桥

若删除**无向连通图**中的一条边, 使得整个图不连通, 那么称这条边为桥。

同样维护 dfn 与 low, 在 DFS 树上处理 $u \to v$ 的过程中, (u,v) 为桥当且仅当 (u,v) 无重边且 $dfn[u]$ **小于**$low[v]$。

至于判断无重边的情况, 可在 DFS 过程中让其儿子返回是否存在重边。

```
0 struct EdgeT {
    int u,v;
    EdgeT(int u,int v):u(u),v(v) {}
} ET[maxm];
int dfn[size], low[size], ccnt = 0, ecnt = 0;
bool DFS(int u, int p, int e) {
    static int icnt = 0;
    dfn[u] = low[u] = ++icnt;
    int pcnt = 0;
    for(int i = last[u]; i; i = E[i].nxt) {
10      int v = E[i].to;
        if(v != p) {
            if(dfn[v])
                low[u] = std::min(low[u], dfn[v]);
            else {
                bool flag = DFS(v, u, i);
                low[u] = std::min(low[u], low[v]);
                if(flag && dfn[u] < low[v])
                    ET[++ecnt] = EdgeT(u, v);
            }
20      }
        else ++pcnt;
    }
    return pcnt==1;
}
```

14.2 强连通分量

14.2.1 定义

强连通 如果一对点存在路径互相可达, 就称这对点强连通。

强连通子图 强连通子图的点两两互相可达。

强连通分量 有向图的极大强连通子图。
一般可以将强连通分量缩成一个点, 然后对缩点后的图进行 dp。

14.2.2 Tarjan 算法

Tarjan 算法求 SCC 的步骤如下:

1. DFS 遍历每个未遍历的点。
2. 对于每个点, 维护其访问时间 dfn, 可访问到的栈上的最早的点的访问时间 low。DFS 处理该点时, 将该点加入栈中。
3. 若 $dfn[u] == low[u]$, 则说明栈上从栈顶到自己的点构成了强连通分量, 新建一个强连通分量, 记录每个点所属的强连通分量。

正确性证明留坑待补。
代码如下:

```
int dfn[size],low[size],st[size],top=0,col[size],ccnt=0;
bool flag[size];
void tarjan(int u) {
    static int icnt=0;
    dfn[u]=low[u]=++icnt;
    flag[u]=true;
    st[++top]=u;
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(dfn[v]) {
            if(flag[v])
                low[u]=std::min(low[u],dfn[v]);
        }
        else {
            DFS(v);
            low[u]=std::min(low[u],low[v]);
        }
    }
    if(dfn[u]==low[u]) {
        int c=++ccnt,v;
        do {
            v=st[top--];
            flag[v]=false;
            col[v]=c;
        } while(u!=v);
    }
}
for(int i=1;i<=n;++i)
    if(!dfn[i])
        DFS(i);
```

14.3 双连通分量

14.3.1 点双连通分量

如果连通图中任意两点之间至少存在两条点不重复路径, 则称该图为点双连通的。无向图中点双连通的极大子图称为点双连通分量。

可以发现当找到一个割点时, 就已经访问了一个点双连通分量, 所以在 Tarjan 算法求割点的基础上记录栈中元素就能得到点双。注意一个割点可能属于多个点双, 因此割点的编号是无意义的, 并且需要记录的元素是边而不是点。

```
int dfn[size], low[size], st[size], top = 0, col[size],
    ccnt = 0;
std::vector<int> bcc[size];
void DFS(int u, int p) {
    static int icnt = 0;
    dfn[u] = low[u] = ++icnt;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(dfn[v]) {
            if(dfn[v] < dfn[u] && v != p) {
                st[++top] = i;
                low[u] = std::min(low[u], dfn[v]);
            }
        }
        else {
            st[++top] = i;
            DFS(v, u);
            low[u] = std::min(low[u], low[v]);
            if(dfn[u] <= low[v]) {
                int c = ++ccnt,eid;
                do {
                    eid = st[top--];
                    int eu = E[eid ^ 1].to;
                    int ev = E[eid].to;
                    if(col[eu] != c)
                        bcc[c].push_back(eu);
                    if(col[ev]!=c)
                        bcc[c].push_back(ev);
                    col[eu] = col[ev] = c;
                } while(eid != i);
            }
        }
    }
}
```

14.3.2 边双连通分量

如果连通图中任意两点之间至少存在两条边不重复路径, 则称该图为边双连通的。无向图中边双连通的极大子图称为边双连通分量。

把桥删去后分出的连通块就是边双连通分量，可以一遍 DFS 求出桥，再一遍 DFS 染色。

上述内容参考了 vufw_795 的博客[1]。

14.4 2-SAT

14.4.1 问题描述

有若干个布尔变量，对于由多个 AND 连接的若干个 OR 子表达式，且子表达式的操作数为布尔变量或者其否定，判断是否存在一组对布尔变量的赋值，使得这个布尔表达式的值为 1。

这类问题一般描述为一个变量为真/假限制了另一个变量必须为真/假，求使得所有限制被满足的一组变量赋值方案。对每个布尔变量拆点，可以将这些限制表示为有向图。将命题中的条件向结论连边，同时对逆否命题的条件和结论连边。

比如要求变量 X 为真时变量 Y 必定为真，首先连边 $X_1 \to Y_1$；其逆否命题为 Y 为假时 X 必定为假，连边 $Y_0 \to X_0$。

对于强制某一变量为真/假的需求，独立出来不好做，考虑沿用连边的思路。如果强制变量 X 为真，就连边 $X_0 \to X_1$。

14.4.2 可行性判定

对这个图求强连通分量，发现同一强连通分量的点同时被选或不被选（分量里有一个点被选，根据边的意义和强连通分量的定义，它可以把这个强连通分量内的点染色为被选）。

因此可以对该图求强连通分量，若 X_0 与 X_1 在同一个强连通分量内则为无解。

14.4.3 构造方案

以下方法的正确性证明留坑待补。

14.4.3.1 DFS 染色法

首先对其进行缩点，标记每个强连通分量的对立分量，并连反图。

按照拓扑序处理每个强连通分量，DFS 对立分量：

1. 若自己已被标记则返回；
2. 将自己标记为不选择，将对立分量标记为选择；
3. DFS 递归标记所连的点。

查看每个点所在强连通分量的标记来输出方案。

14.4.3.2 更简洁的方法

由于 tarjan 算法标记强连通分量的顺序为自底向上，而上述方法的顺序同样也是自底向上。因此对于每个布尔变量，选取所在强连通分量标号小的布尔值。

该方法参考了 TRTTG 的博客[2]。

由于 01 变量的染色过程是连续的，因此当 X_0 与 X_1 较后一个的颜色确定后就可以进行判断。这种方式可以避免在非法解上浪费时间。判断过程要放在 if(dfn[i]) 外!!!

[1] Tarjan 三大算法之双连通分量（双连通分量）https://blog.csdn.net/fuyukai/article/details/51303292

[2] 2-SAT 问题的方案输出 https://www.cnblogs.com/TheRoadToTheGold/p/8436948.html#_label1

14.4.4 前后缀优化建图

例题 PA2010 Riddle

有 n 个城镇, 隶属于 k 个郡, 城镇之间有 m 条连边。现在要将这些城镇中设立首都, 满足任意一条边之间至少有一个城镇为首都, 且每个郡最多有一个首都, 求是否存在合法方案。

点覆盖的约束很好解决, 但是限制每个郡最多有一个首都比较困难, 因为常规思路的连边达到 $O(n^2)$ 级别。考虑如何每个点只连常数条边就可以把不选的约束传递到同一个郡内的所有节点。记布尔变量 x_i 表示城镇 i 是否为首都, 记布尔变量 x'_i 表示在其所在郡的城镇序列中, 城镇 i 及其之前的城镇是否被选。根据单点与前缀、前缀与前缀的关系连边, 就可以把约束快速传递。由于前缀与前缀之间的约束关系可以将约束后传, 不必对后缀连边。

该方法参考了 ZigZagK 的博客[3]。

14.5 仙人掌与圆方树

14.5.1 仙人掌

仙人掌 任意一条边最多只存在于一个环中的无向连通图叫做仙人掌。

14.5.1.1 仙人掌的判定

假设整个图已经连通, 可以先对仙人掌进行 DFS, 记录 DFS 序与点的深度。然后树上差分求出经过这条边的环数。判定的同时还可以把仙人掌去环变成森林。

```
0  void initGraph(int n) {
       memset(last, n);
       cnt = 1;
   }
   bool flag[size];
   int p[size], d[size], id[size], icnt;
   void DFS(int u) {
       id[++icnt] = u;
       flag[u] = true;
       for(int i = last[u]; i; i = E[i].nxt) {
10         int v = E[i].to;
           if(!flag[v]) {
               d[v] = d[u] + 1;
               p[v] = u;
               DFS(v);
           }
       }
   }
   int tag[size];
   bool graph2Forest(int n) {
20     icnt = 0;
       memset(flag, n);
       DFS(1);
```

[3]【前后缀优化建图 +2-SAT】BZOJ3495(PA2010)[Riddle] 题解
https://blog.csdn.net/zzkksunboy/article/details/76285426

```
    if(icnt != n)
        return false;
    memset(tag, n);
    for(int i = 2; i <= cnt; i += 2) {
        int u = E[i].to, v = E[i ^ 1].to;
        if(d[u] < d[v])
            std::swap(u, v);
        if(p[u] != v)
            ++tag[u], --tag[v];
    }
    initGraph(n);
    for(int i = n; i >= 1; --i) {
        int u = id[i];
        tag[p[u]] += tag[u];
        if(tag[u] == 0) {
            if(p[u]) {
                addEdge(u, p[u]);
                addEdge(p[u], u);
            }
        } else if(tag[u] > 1)
            return false;
    }
    return true;
}
```

如果 graph2Forest 返回 false 则说明这个图不是仙人掌。若返回 true 则建出一个去掉环的森林。

14.5.1.2 DFS 树 dp 法

对于简单的仙人掌问题可以使用 DFS 树做法:

首先可以使用类似 Tarjan 的算法判断是否出现了环, 然后对于环和桥分别 dp, 将环的信息记录在环在 DFS 树上最浅的节点上(即在回到环上最浅点时另外 dp), 这样向上转移时就和桥一样了。

模板:

```
int d[size], p[size], dfn[size], low[size], icnt = 0;
void DFS(int u) {
    dfn[u] = low[u] = ++icnt;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v == p[u])
            continue;
        if(!dfn[v]) {
            p[v] = u;
            d[v] = d[u] + 1;
            DFS(v);
        }
        low[u] = std::min(low[u], low[v]);
        if(low[v] > dfn[u])
            //转移桥边 v->u
```

```
    }
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(p[v] != u && dfn[u] < dfn[v])
            solveRing(u, v);//计算环 v->u
    }
}
```

14.5.2 圆方树

圆方树是解决仙人掌问题的利器, 主要思想是把仙人掌构造为一棵树, 然后使用熟悉的树上操作来处理。

14.5.2.1 构造

在 Tarjan 时连树边(圆圆边), 在环上深度最浅的点上处理环, 把环上的节点(圆点)都连到新建的点(方点)上(圆方边)。

```
int dfn[size],low[size],d[size],p[size],ncnt;
void solveRing(int u,int v) {
    int siz=d[v]-d[u]+1;
    int id=++ncnt;
    for(int i=v;i!=u;i=p[i])
        //add id->i
    //add u->id
}
void tarjan(int u) {
    static int icnt=0;
    dfn[u]=low[u]=++icnt;
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(v==p[u])
            continue;
        if(!dfn[v]) {
            p[v]=u;
            d[v]=d[u]+1;
            DFS(v);
            low[u]=std::min(low[u],low[v]);
        }
        else low[u]=std::min(low[u],dfn[v]);
        if(dfn[u]<low[v])
            //add u->v
    }
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(v!=p[u] && p[v]!=u && dfn[u]<dfn[v])
            solveRing(u,v);
    }
}
```

```
ncnt=n;
```

还有一种简洁的建树方法:

```
int dfn[size],p[size],icnt=0,ncnt;
bool ring[size];
void DFS(int u) {
    dfn[u]=++icnt;
    for(int i=last[u];i;i=E[i].nxt) {
        int v=E[i].to;
        if(v==p[u])
            continue;
        if(dfn[v]) {
            if(dfn[v]<dfn[u]) {
                int id=++ncnt;
                for(int j=u;j!=v;j=p[j])
                    addEdge(id,j),ring[j]=true;
                addEdge(v,id),ring[v]=true;
            }
        }
        else {
            p[v]=u;
            ring[u]=false;
            DFS(v);
            if(!ring[u])
                addEdge(u,v);
        }
    }
}
ncnt=n;
```

参见 Iking123 的博客[4]。

14.5.2.2 性质

子仙人掌 以 r 为根的仙人掌上点 p 的子仙人掌为去掉 p 到 r 的简单路径后点 p 所在的连通块。

性质 14.1 以 r 为根仙人掌上点 p 的子仙人掌对应圆方树上点 p 的子树。

性质 14.2 圆方树不存在方点与方点的连边。

14.5.2.3 应用

最短路 把圆圆边的权值设为原边权值, 圆方边的权值设为当前点到方点父亲的最短距离(可以维护到树根的距离来计算), 方圆边的权值设为 0。

类比树上距离的做法, 树链剖分后对每个询问求 LCA。

- 若 LCA 为圆点, 则按照树上距离的方法解决。

[4]圆方树/广义圆方树学习小记(gradually update...)
https://blog.csdn.net/qq_36551189/article/details/81047872

- 若 LCA 为方点, 则说明该路径经过了环上的两个点, 但走哪一侧还未知。首先计算询问的这两个点分别在哪棵子树中, 计算两条路径的答案, 接下来只要考虑环上路径。询问子树时需要对在 LCA 环上的祖先是否为重儿子进行分类, jump 函数计算该点的编号:

```
0     int jump(int u,int lca) {
          int res;
          while(top[u]!=top[lca]) {
              res=top[u];
              u=p[top[u]];
          }
          return u==lca?res:son[lca];
      }
```

预先在计算方圆边距离时标记走的方向, 然后分类讨论计算环上两点的最短路。

如此可以保证经过环的时候走的是最短路。

点分治 注意处理方点时的复杂度, 一般将方点设为环的大小, 点分治时找带权重心。

14.5.3 广义圆方树

对于每个点双, 将点双内的点(称为圆点)连到新点(称为方点)。

```
0 int dfn[size], low[size], st[size], timeStamp = 0,
    top = 0, nsiz;
  void tarjan(int u) {
      dfn[u] = low[u] = ++timeStamp;
      st[++top] = u;
      for(int i = g1.last[u]; i; i = g1.E[i].nxt) {
          int v = g1.E[i].to;
          if(dfn[v])
              low[u] = std::min(low[u], dfn[v]);
          else {
10            tarjan(v);
              low[u] = std::min(low[u], low[v]);
              if(dfn[u] <= low[v]) {
                  int s = ++nsiz, p;
                  g2.addEdge(u, s);
                  do {
                      p = st[top--];
                      g2.addEdge(s, p);
                  } while(p != v);
              }
20        }
      }
  }
  nsiz = n;
```

广义圆方树的点与边数的最大规模都是 $2V$ 级别的(圆点与方点相间, 并且连成一棵树)。

关键在于考虑圆点和方点的权值和边权, 注意更新方点时不要计算父亲的贡献, 仅计算子树的贡献, 在另外的计算过程中考虑父亲的贡献, 以保证更新的复杂度。

上述内容参考了小蒟蒻 yyb[5]和 immortalCO[6]的博客。

14.6 图上路径

14.6.1 欧拉路径与欧拉回路

这两种路径都要求经过每条边有且只有一次, 后者要求起点和终点相同。

14.6.1.1 无向图欧拉路径

判定 无向图欧拉路径的奇点个数为 0(欧拉回路)或 2。这里使用 Hierholzer 算法, 时间复杂度 $O(E)$。

首先统计奇点个数, 若奇点个数为 0 则任意选取点作为起点, 奇点个数为 2 则任意选取奇点之一作为起点。

接下来从起点开始 DFS:

1. 选取与点 u 相连的边 (u,v), 删除 $(u,v),(v,u)$, 然后 $DFS(v)$。
2. 向队列中加入点 u。

遍历边时可以使用类似于 Dinic 的当前弧优化, 若要求字典序最小, 使用 $std::multiset$ 存边。

14.6.1.2 有向图欧拉回路

有向图欧拉路径有 0(欧拉回路)或 2 个点不满足入度 = 出度的要求。然后继续使用 Hierholzer 算法, 注意输出时要反向(因为 DFS 是反向放入队列的)。

板子(UOJ117):

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005;
struct Edge {
    int to, nxt;
} E[size * 4];
int last[size], cnt;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
```

[5]仙人掌 & 圆方树学习笔记 https://www.cnblogs.com/cjyyb/p/9098400.html
[6]圆方树——处理仙人掌的利器 http://immortalco.blog.uoj.ac/blog/1955

```
    last[u] = cnt;
}
int q[size * 2], qcnt = 0;
void DFS1(int u) {
    for(int& i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v) {
            E[i].to = E[i ^ 1].to = 0;
            int eid = i;
            DFS1(v);
            int reid = eid >> 1;
            q[++qcnt] = (eid & 1 ? reid : -reid);
        }
    }
}
int in[size], out[size];
bool foo1(int n, int m) {
    cnt = 1;
    for(int i = 0; i < m; ++i) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
        ++in[v], ++in[u];
    }
    for(int i = 1; i <= n; ++i)
        if(in[i] & 1)
            return false;
    for(int i = 1; i <= n; ++i)
        if(in[i]) {
            DFS1(i);
            break;
        }
    if(qcnt != m)
        return false;
    puts("YES");
    for(int i = 1; i <= m; ++i)
        printf("%d ", q[i]);
    return true;
}
void DFS2(int u) {
    for(int& i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v) {
            E[i].to = 0;
            int eid = i;
            DFS2(v);
            q[++qcnt] = eid;
        }
    }
```

```
}
bool foo2(int n, int m) {
    cnt = 0;
    for(int i = 0; i < m; ++i) {
        int u = read();
        int v = read();
        addEdge(u, v);
        ++out[u], ++in[v];
    }
    for(int i = 1; i <= n; ++i)
        if(in[i] != out[i])
            return false;
    for(int i = 1; i <= n; ++i)
        if(in[i]) {
            DFS2(i);
            break;
        }
    if(qcnt != m)
        return false;
    puts("YES");
    for(int i = m; i >= 1; --i)
        printf("%d ", q[i]);
    return true;
}
int main() {
    int t = read();
    int n = read();
    int m = read();
    if(!(t == 1 ? foo1(n, m) : foo2(n, m)))
        puts("NO");
    puts("");
    return 0;
}
```

14.6.1.3 混合图欧拉回路

对于既有无向边又有有向边的图, 主要思路是对无向边进行定向, 使其满足入度 = 出度, 然后转换为有向边欧拉回路问题来做。

首先判断图的连通性, 对无向边进行随机定向, 把图转换为有向图, 然后统计每个点入度与出度, 若入度与出度奇偶性不同则无解(每次调整方向都会导致它们的差变动 2, 无法调整至相等)。

接下来利用网络流的自动调整功能, 从超级源向每个入度 < 出度的点连流量为 $\frac{出度-入度}{2}$ 的边, 从每个入度 > 出度的点向超级汇连流量为 $\frac{入度-出度}{2}$ 的边, 对于原图中定向为 $u \to v$ 的边从 u 到 v 连一条流量为 1 的边。最后跑最大流, 若满流则说明有解。无向边对应网络流的双边中残余流量非 0 的边就是该无向边定向后的边。此时满足每个点的入度 = 出度, 跑有向边欧拉回路求得方案。

正确性证明 每次对一条增广路进行增广时, 相当于把路径上所有的边反向, 路径中间的点的入度和出度保持不变, 而起点的入度 +1, 出度-1, 终点的入度-1, 出度 +1。如果最大流

满流, 则所有的点都已经满足入度 = 出度了。此时如果这条边有流量, 说明它所对应的无向边应该定向为自己的反向。

以上内容参考了 ajcxsu[7]和 commonc[8]的博客。

14.6.2 哈密尔顿回路

哈密尔顿路径要求经过每个点有且只有一次, 哈密尔顿回路还要求起点和终点相同。

14.6.2.1 判定

定理 14.3 任意竞赛图存在哈密尔顿路径, 若此图强连通则存在哈密尔顿回路。

构造只能使用状压 DP$O(2^n \cdot n^3)$ 解决。

14.6.2.2 旅行商问题 TSP

旅行商问题求的是无向完全图的最短哈密尔顿回路, 是 NP 问题。

若距离函数满足三角不等式(比如欧几里得距离), 存在其解不超过最优解两倍的近似算法: 对该图求 MST, DFS 该 MST, 遍历的路径去掉重复点后就是其解。时间复杂度 $O(V^2)$。

还存在其解不超过最优解 $\frac{3}{2}$ 的近似算法(Christofides 算法):

- 求该图的 MST;
- 对 MST 中度数为奇数的点做无向图最小权完美匹配, 将匹配边加入 MST;
- 求出欧拉回路;
- 按照欧拉回路走, 跳过重复节点。

时间复杂度 $O(V^3)$。

该方法参考了 Wikipedia-EN[9]。

若不满足三角不等式, 可以使用 $O(V^2)$ 的最近邻算法, 即每次 DFS 都贪心地选取最短的边递归。

14.7 弦图

以下所述的图均为无向图。

14.7.1 相关概念

14.7.1.1 团

图 G 的子图 $G' = (V', E')$, G' 是 V' 的完全图。

14.7.1.2 极大团

一个不是其它团的子集的团。

[7][模板][持续更新] 欧拉回路与欧拉路径浅析 https://www.cnblogs.com/acxblog/p/7390301.html
[8]混合图的欧拉回路 https://blog.csdn.net/commonc/article/details/52442882
[9]Christofides algorithm - Wikipedia
https://en.wikipedia.org/wiki/Christofides_algorithm

14.7.1.3 最大团

点数最大的团。记最大团的大小为团数 $\omega(G)$。

14.7.1.4 最小染色

色数 $\chi(G)$ 为使得相邻点颜色不同的最小颜色数。

引理 14.4 $\omega(G) \leq \chi(G)$

14.7.1.5 弦

连接环中不相邻两点的边。

14.7.1.6 弦图

图中任意长度大于 3 的环都至少有一根弦。

14.7.1.7 诱导子图

诱导子图 $G' = (V', E')$, 其中 $V' \subseteq V, E' = \{(u, v)|u, v \in V', (u, v) \in E\}$。

引理 14.5 弦图的诱导子图仍然是弦图。

14.7.2 弦图判定

14.7.2.1 单纯点

点 u 为单纯点当且仅当点 u 以及其邻接点构成的诱导子图是一个团。

引理 14.6 弦图至少有一个单纯点,若其不为完全图则至少有两个不相邻的单纯点。

14.7.2.2 完美消除序列

点集的序列 $v_1, v_2, \cdots, v_n$ 是完美消除序列当且仅当 v_i 在 $\{v_i, v_{i+1}, \cdots, v_n\}$ 的诱导子图上是单纯点。

定理 14.7 图 G 是弦图当且仅当其存在完美消除序列。

充分性: 使用数学归纳法, 假设弦图的诱导子图有完美消除序列, 由上文两个引理得该弦图的完美消除序列可由单纯点 $u+$ 剩余点的诱导子图的完美消除序列得到。

必要性: 设出现在完美消除序列中的某点为 u, 根据完美消除序列的定义得与 u 相连的所有点中点对之间有连边, 故不存在长度 > 3 的无弦环。

14.7.2.3 朴素判定算法

每次找到一个单纯点 v 并加入完美消除序列中, 然后删除 v 及其连边。直到所有点都被删除或找不到单纯点为止。时间复杂度 $O(n^4)$。

14.7.2.4 最大势算法

维护每个点是否被标号以及相邻标号点的数量。每次选择未被标号且相邻已标号点数量最多的点标号。序列顺序与标号顺序相反。使用链表而不是优先队列实现（类似于HLPP），时空复杂度为 $O(n+m)$。

定理 14.8 若该图是弦图，则最大势算法生成的是完美消除序列。

证明留坑待补。

接下来要判断这个序列是否为完美消除序列。设 v_i 与 $v_{i+1}, v_{i+2}, \cdots, v_n$ 中的 $u_1, u_2, \cdots, u_k$ 相邻，仅需检查 u_1 是否与 $u_2, \cdots, u_k$ 是否全相邻（若全相邻则不仅保证了 u_1 到 $u_2, \cdots, u_k$ 的相邻，还会触发 u_2 到 $u_3, \cdots, u_k$ 的检查，以此类推可以遍历到整个团的边）。时间复杂度 $O(n+m)$。

模板（SP5446 FISHNET - Fishing Net）：

```
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1005;
struct Graph {
    struct Edge {
        int to, nxt;
    } E[size * size];
    int last[size], cnt;
    void reset(int n) {
        cnt = 0;
        memset(last, 0, sizeof(int) * (n + 1));
    }
    void addEdge(int u, int v) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u];
        last[u] = cnt;
    }
} G, L;
bool link[size][size];
int id[size], label[size];
bool foo(int n, int m) {
    G.reset(n);
    L.reset(n);
    memset(link, 0, sizeof(link));
    while(m--) {
```

```
        int u = read();
        int v = read();
        G.addEdge(u, v);
        G.addEdge(v, u);
        link[u][v] = link[v][u] = true;
    }
    for(int i = 1; i <= n; ++i)
        L.addEdge(0, i);
    memset(label + 1, 0, sizeof(int) * n);
    memset(id + 1, 0, sizeof(int) * n);
    int highest = 0, cid = n;
    while(highest != -1 && cid) {
        if(L.last[highest] == 0) {
            --highest;
            continue;
        }
        for(int& i = L.last[highest]; i;
            i = L.E[i].nxt) {
            int u = L.E[i].to;
            if(!id[u]) {
                id[u] = cid;
                --cid;
                for(int i = G.last[u]; i;
                    i = G.E[i].nxt) {
                    int v = G.E[i].to;
                    if(!id[v]) {
                        L.addEdge(++label[v], v);
                        if(label[v] > highest)
                            highest = label[v];
                    }
                }
                break;
            }
        }
    }
    id[0] = 1 << 30;
    for(int u = 1; u <= n; ++u) {
        int v0 = 0;
        for(int j = G.last[u]; j; j = G.E[j].nxt) {
            int v = G.E[j].to;
            if(id[v] > id[u] && id[v] < id[v0])
                v0 = v;
        }
        if(v0) {
            for(int j = G.last[u]; j; j = G.E[j].nxt) {
                int v = G.E[j].to;
                if(id[v] > id[v0] && !link[v0][v])
                    return false;
            }
        }
```

```
    }
    return true;
}
int main() {
    int n, m;
    while(scanf("%d%d", &n, &m) != EOF) {
        if(n == 0 && m == 0)
            break;
        puts(foo(n, m) ? "Perfect" : "Imperfect");
    }
    return 0;
}
```

14.7.3 弦图的极大团

定理 14.9 弦图的极大团一定是某个点 u_i 及其在完美消除序列的后缀 $u_{i+1}, \cdots, u_n$ 上的邻接点构成的诱导子图。

证明:

设弦图的某个极大团的点集为 V, 记 v 为点集 V 中在完美消除序列最前端的点, V' 为 v 及其序列后缀邻接点组成的集合, 有 $V \subseteq V'$。又因为集合 V' 的诱导子图是团, 而 V 的诱导子图是极大团, 所以 $V = V'$。

推论 14.10 弦图最多有 n 个极大团。

接下来仅需枚举每个点, 查看其对应诱导子图是否为极大团。统计每个节点的后缀邻接点数 C, 记点 u 的第一后缀邻接点为 v, 点 v 对应的团不是极大团当且仅当 $C_v + 1 \leq C_u$。

14.7.4 弦图的点染色

即求弦图的色数 $\chi(G)$。

定理 14.11 团数 $=$ 色数

证明 求出完美消除序列后从后往前贪心染色, 事实上由于完美消除序列的性质, 节点 u 的染色编号为 $C_u + 1$。这里求出的颜色数实际上为弦图的团数。又因为团数 $\leq$ 色数, 而求出的颜色数 $\geq$ 色数, 所以团数 $=$ 色数。

模板(HNOI2008 神奇的国度):

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
```

```
const int size = 10005;
struct Graph {
    struct Edge {
        int to, nxt;
    } E[size * 200];
    int last[size], cnt;
    Graph() : cnt(0) {}
    void addEdge(int u, int v) {
        ++cnt;
        E[cnt].to = v, E[cnt].nxt = last[u];
        last[u] = cnt;
    }
} G, L;
int id[size], label[size];
int main() {
    int n = read();
    int m = read();
    while(m--) {
        int u = read();
        int v = read();
        G.addEdge(u, v);
        G.addEdge(v, u);
    }
    for(int i = 1; i <= n; ++i)
        L.addEdge(0, i);
    int highest = 0, cid = n;
    while(highest != -1 && cid) {
        if(L.last[highest] == 0) {
            --highest;
            continue;
        }
        for(int& i = L.last[highest]; i;
            i = L.E[i].nxt) {
            int u = L.E[i].to;
            if(!id[u]) {
                id[u] = cid--;
                for(int i = G.last[u]; i;
                    i = G.E[i].nxt) {
                    int v = G.E[i].to;
                    if(!id[v]) {
                        L.addEdge(++label[v], v);
                        if(label[v] > highest)
                            highest = label[v];
                    }
                }
                break;
            }
        }
    }
    int res = 0;
```

```
    for(int i = 1; i <= n; ++i) {
        int cnt = 1;
        for(int j = G.last[i]; j; j = G.E[j].nxt) {
            int v = G.E[j].to;
            cnt += id[v] > id[i];
        }
        if(cnt > res)
            res = cnt;
70  }
    printf("%d\n", res);
    return 0;
}
```

14.7.5 弦图的最大独立集与最小团覆盖

求出完美消除序列后从前往后贪心选择节点。

定理 14.12 最大独立集 = 最小团覆盖

最小团覆盖即为最大独立集中的每个点对应点集的并。
证明留坑待补。

14.7.6 区间图

给定若干个区间, 将每个区间当做一个点, 两点有边当且仅当两区间有交集, 这样的图称为区间图。区间图也是弦图, 因为区间图不存在长度 >3 的无弦环。

最大不重叠区间 该题原有贪心解法。可以建出区间图后求弦图最大独立集。当然这题有贪心解法意味着我们不必建出实际的区间图, 可以直接使用贪心解法生成一个完美消除序列:按照右端点升序排序。

积木下落问题 有一堆积木, 每个积木有一个起始释放的水平坐标范围, 它们的高度均为 1, 选择积木下落顺序使得积木总高度最小。
这其实是区间图最小染色问题, 因为区间相交的积木不可以同时下落。
上述内容参考了陈丹琦在 WC2009 上的讲稿《弦图与区间图》[8]。

14.8 带花树

带花树主要用来解决无向图最大匹配与完美匹配问题。

14.8.1 无向图最大匹配

尝试使用匈牙利算法计算无向图最大匹配, 即每次选取一个未匹配点, 以此为树根, DFS 遍历出交错树。
定义交错树上的 “奇点” 为距树根距离为奇数的点, “偶点” 为距树根距离为偶数的点。
那么对于二分图有如下两种情况:

- 奇点连到偶点:一定走匹配边。

- 偶点连到奇点:一定走未匹配边。

但是对于无向图来说有一点不同:

- 奇点连到偶点:一定走匹配边。
- 偶点连到奇点:一定走未匹配边。
- 偶点连到偶点:一定走未匹配边。此时我们把交错树上的两个偶点的交错路径以及偶点到偶点的边形成的奇环称为“花”,树根称为“花托”。

考虑跨过偶点到偶点的边,可以发现原先的奇点可以变为偶点,然后可以走未匹配边延伸出更多的交错路径。既然花上的点都可以是偶点,不妨将其直接缩为一个偶点,这就是“缩花”。一朵花至少有 3 个点,因此最多缩花 $\frac{|V|}{2}$ 次,这里使用并查集缩花。

其余步骤与匈牙利算法类似:

遍历未匹配点尝试建立交错树以寻找增广路径:

使用 BFS 遍历,队列仅维护偶点。

- 若走到未访问点:
  - 若该点已匹配:将该点加入队列后退出,继续遍历交错树。
  - 若该点未匹配:找到增广路径,翻转链上边的匹配状态后返回 true。
- 若走到已访问点:
  - 若该点为奇点:由于该点已访问,什么也不要做。
  - 若该点为偶点:缩花,参数为两个偶点标号 u, v。

缩花:

1. 使用 DSU 找出花托,即 $LCA(u, v)$。
2. 两遍调用缩花的一边并把奇点标记为偶点,加入队列。

假设并查集复杂度为 $O(1)$,算法时间复杂度 $O(|V||E|)$。二分图最大匹配中的 Greedy Matching 仍然适用。

模板:

```
#include <algorithm>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 505;
struct Edge {
```

```
    int to, nxt;
} E[size * size];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int fa[size];
int find(int u) {
    return fa[u] ? fa[u] = find(fa[u]) : u;
}
int match[size], q[size * size], d[size], pre[size],
    cts = 0, ts[size];
int getLCA(int u, int v) {
    ++cts;
    while(true) {
        if(u) {
            u = find(u);
            if(ts[u] == cts)
                return u;
            else
                ts[u] = cts, u = pre[match[u]];
        }
        std::swap(u, v);
    }
}
void contract(int u, int v, int p, int& e) {
    while(find(u) != p) {
        pre[u] = v;
        v = match[u];
        if(d[v] == 2)
            d[v] = 1, q[e++] = v;
        if(fa[u] == 0)
            fa[u] = p;
        if(fa[v] == 0)
            fa[v] = p;
        u = pre[v];
    }
}
bool aug(int s, int n) {
    memset(fa + 1, 0, sizeof(int) * n);
    memset(d + 1, 0, sizeof(int) * n);
    memset(pre + 1, 0, sizeof(int) * n);
    q[0] = s, d[s] = 1;
    int b = 0, e = 1;
    while(b < e) {
        int u = q[b++];
        for(int i = last[u]; i; i = E[i].nxt) {
            int v = E[i].to;
```

```
            if(find(u) == find(v))
                continue;
            if(d[v] == 1) {
                int lca = getLCA(u, v);
                contract(u, v, lca, e);
                contract(v, u, lca, e);
            } else if(d[v] == 0) {
                d[v] = 2, pre[v] = u;
                if(!match[v]) {
                    int cp = v;
                    while(cp) {
                        int p = pre[cp];
                        match[cp] = p;
                        std::swap(cp, match[p]);
                    }
                    return true;
                }
                d[match[v]] = 1, q[e++] = match[v];
            }
        }
    }
    return false;
}
int main() {
    int n = read();
    int m = read();
    while(m--) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    int res = 0;
    for(int i = 1; i <= n; ++i)
        if(!match[i])
            for(int j = last[i]; j; j = E[j].nxt) {
                int v = E[j].to;
                if(!match[v]) {
                    match[v] = i, match[i] = v, ++res;
                    break;
                }
            }
    for(int i = 1; i <= n; ++i)
        if(match[i] == 0 && aug(i, n))
            ++res;
    printf("%d\n", res);
    for(int i = 1; i <= n; ++i)
        printf("%d ", match[i]);
    return 0;
}
```

14.8.2 Micali-Vazirani Algorithm

该算法也用于求无向图最大匹配, 时间复杂度 $O(\sqrt{|V|}|E|)$。

由于找不到中文资料, 暂时先坑着。参考 Silvio Micali 与 Vijay V. Vazirani 的论文 [9]。

14.8.3 近线性复杂度的随机匹配算法

这里介绍的是 2011 年由 Anant Jindal, Gazal Kochar, Manjish Pal 提出的基于马尔科夫链和 Glauber Dynamics 的随机算法, 参见论文 [10]。

算法步骤如下(调用下述算法 $10 \lg n$ 次):

1. 令 M_0 为任意匹配, $p = 2^{|E|}$。
2. 迭代 $k = 10|E| \lg |V|$ 次更新 M_i。
   (a) 随机取边 e;
   (b) 令 M' 有 $\frac{p}{1+p}$ 的概率为 $M_i \cup \{e\}$, 有 $\frac{1}{1+p}$ 的概率为 $M_i \backslash \{e\}$。
   (c) 若 M' 为匹配则令 $M_{i+1} = M'$, 否则 $M_{i+1} = M_i$。
3. 输出 M_k。

时间复杂度 $O(|E| \lg^2 |V|)$, 算法正确率为 $1 - \left(\frac{169}{189}\right)^{10 \lg n}$。

模板(不正确, 待修复):

为了防止错过最优解, 我还加入了备份操作, 最坏时间复杂度 $O(|V|^2)$。

```
#include <chrono>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <limits>
#include <random>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 505;
struct Edge {
    int u, v;
} E[130005];
int match[size], bak[size], bv = 0;
void save(int n, int cur) {
    if(cur > bv) {
        bv = cur;
        memcpy(bak + 1, match + 1, sizeof(int) * n);
```

```
    }
}
using REng = std::mt19937_64;
using Clock = std::chrono::high_resolution_clock;
using IntType = REng::result_type;
int main() {
    int n = read();
    int m = read();
    for(int i = 0; i < m; ++i) {
        E[i].u = read();
        E[i].v = read();
    }
    REng rnd(Clock::now().time_since_epoch().count());
    std::uniform_int_distribution<> uid(0, m - 1);
    int mink = ceil(log(1e-5) / log(169.0 / 189.0));
    int k1 = 10 * ceil(log2(n)),
        ka = std::max(k1, mink), mp = n >> 1;
    IntType p = std::numeric_limits<IntType>::max() /
        ((IntType(1) << m) + 1);
    if(IntType(1) << m == 0)
        p = 1;
    while(ka--) {
        memset(match + 1, 0, sizeof(int) * n);
        int k = k1 * m, cur = 0;
        while(k--) {
            int eid = uid(rnd);
            int u = E[eid].u, v = E[eid].v;
            if(rnd() <= p) {
                if(match[u] == v) {
                    save(n, cur);
                    match[u] = match[v] = 0;
                    --cur;
                }
            } else {
                if(match[u] == 0 && match[v] == 0) {
                    match[u] = v, match[v] = u;
                    ++cur;
                    if(cur == mp)
                        break;
                }
            }
        }
        save(n, cur);
        if(bv == mp)
            break;
    }
    printf("%d\n", bv);
    for(int i = 1; i <= n; ++i)
        printf("%d ", bak[i]);
    return 0;
```

}

14.8.4 无向图最大权匹配

似乎并没有比较短的实现。。。暂时弃疗。
上述内容参考了江任捷的演算法筆記[10]、permui[11]和 hcsoso[12]的博客。

14.9 支配树

14.9.1 定义

支配树基于某个原点 s, 若 s 到某个点 v 的节点必定经过 u, 则称 u 支配 v。起点 s 与自己 v 称为平凡支配点。

若 v 的某个支配点 w 满足其被 v 的其余非平凡支配点支配, 则 w 为 v 的最近支配点, 记作 $idom(v) = w$。每个点到自己的 $idom$ 连边, 则构造出了一棵树, 称为支配树。

14.9.2 DAG 的支配树

DAG 的支配树构造较为简单。考虑按照 top 序加入节点, 当前节点的 $idom$ 就是其前驱节点在支配树上的 LCA。由于树的形态是固定的, 使用倍增可以 $O((n+m)\lg n)$ 实现。

14.9.3 一般图的支配树

该内容留坑待补。
上述内容参考了 MoebiusMeow 的博客[13]。

14.10 杂讲

14.10.1 竞赛图

竞赛图是一个无向完全图被定向后得到的图。

定理 14.13 竞赛图缩点后是一条链。

14.10.1.1 竞赛图判定

竞赛图可以用来指示两两选手比赛的胜负, 判定比分是否合法即判定是否存在合法的竞赛图。

定理 14.14 (Landau's Theorem) 对于一个有序的竞赛图度数序列/得分序列 $0 \le s_1 \le s_2 \le \cdots \le s_n$, 有 $\forall 1 \le k \le n, \sum_{i=1}^{k} s_i \ge \binom{k}{2}$, 当 $k = n$ 时等号必须成立。

对度数/胜利场数排序后逐个判断其合法性。

[10]演算法筆記 - Matching http://www.csie.ntnu.edu.tw/~u91029/Matching.html
[11]一般图最大匹配-带花树算法 https://www.cnblogs.com/owenyu/p/6858508.html
[12]Maximum matching in general graphs https://finiteplayground.wordpress.com/2011/07/15/maximum-matching-in-general-graphs/
[13]康复计划 #4 快速构造支配树的 Lengauer-Tarjan 算法 https://www.cnblogs.com/meowww/archive/2017/02/27/6475952.html

14.10.2 最小平均值环

对于一个有向图, 找出平均值最小的环。

类似于分数规划的思想, 对平均值进行二分, 将所有边权减去二分值, 若存在负环则说明存在环的平均值小于该二分值。

14.10.3 平面图性质

以下仅讨论 $V \geq 3$ 的情况:

性质 14.15 $E \leq 3V - 6$

性质 14.16 $F \leq 2V - 4$

使用这些性质可以限制边数以加速平面图判定。

上述内容参考了 Wikipedia-EN[14]

14.10.4 拓扑排序判环

若拓扑排序无法使得所有点都入队则说明存在环。可以通过枚举点, 将其度数-1 取得删掉一条边的效果。例如 CF915D Almost Acyclic Graph:

```
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 505, esiz = 100005;
struct Edge {
    int to, nxt;
} E[esiz];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int q[size];
bool topSort(int n, int* in) {
    int qcnt = 0;
    for(int i = 1; i <= n; ++i)
        if(!in[i])
```

[14]Planar graph - Wikipedia https://en.wikipedia.org/wiki/Planar_graph

```
            q[++qcnt] = i;
        for(int i = 1; i <= qcnt; ++i) {
            int u = q[i];
            for(int j = last[u]; j; j = E[j].nxt) {
                int v = E[j].to;
                if(--in[v] == 0)
                    q[++qcnt] = v;
            }
        }
        return qcnt == n;
    }
    int in[size], tmp[size];
    bool solve(int n) {
        for(int i = 1; i <= n; ++i)
            if(in[i]) {
                memcpy(tmp + 1, in + 1, sizeof(int) * n);
                --tmp[i];
                if(topSort(n, tmp))
                    return true;
            }
        return false;
    }
    int main() {
        int n = read();
        int m = read();
        for(int i = 1; i <= m; ++i) {
            int u = read();
            int v = read();
            addEdge(u, v);
            ++in[v];
        }
        puts(solve(n) ? "YES" : "NO");
        return 0;
    }
```

14.10.5 Lindström–Gessel–Viennot Lemma

给定一个 DAG, 以及 n 个起点 $a_1, a_2, \cdots, a_n$ 和对应终点 $b_1, b_2, \cdots, b_n$, 求这 n 条点不相交(包括终点)路径的方案数。

根据 Lindström–Gessel–Viennot Lemma, 记 $e(a_i, b_j)$ 为 $a_i \to b_j$ 的路径方案数, 答案为

$$det\left(\begin{bmatrix} e(a_1,b_1) & e(a_1,b_2) & \cdots & e(a_1,b_n) \\ e(a_2,b_1) & e(a_2,b_2) & \cdots & e(a_2,b_n) \\ \vdots & \vdots & \ddots & \vdots \\ e(a_n,b_1) & e(a_n,b_2) & \cdots & e(a_n,b_n) \end{bmatrix}\right)$$

14.10.5.1 推广

实际上 $e(a_i, b_j)$ 为 $a_i \to b_j$ 的所有路径上边权积之和, 类似 Matrix-Tree 定理扩展的讨论(参见第 13.4节)可扩展到边权相关问题。

上述内容参考了 Wikipedia-EN[15]。

14.10.6 三元环计数

给定一个 n 个点 m 条边, 无重边无自环的无向图, 求无序三元组 (i, j, k) 的个数, 其中且两两有边。

算法步骤如下:

- 对无向图的边进行重定向, 度数大的点连向度数小的点, 若度数相同则编号小的点连向编号大的点(反向也可以)。
- 枚举点 u, 将 u 的邻接点标为已访问, 再枚举 u 的邻接点 v, 枚举 v 的邻接点 w, 若 w 被 u 访问, 则 (u, v, w) 是一个合法三元组。

该算法可以保证不重不漏地计数。若 n, m 同阶, 时间复杂度 $O(m\sqrt{m})$。

正确性与时间复杂度参见 KingSann 的博客[16]。

14.10.6.1 竞赛图与三元环

定理 14.17 竞赛图要么是一个拓扑图, 要么存在三元环。

记点数为 n, 点 u 的出度为 out_u, 竞赛图的三元环个数可以由下列公式线性计算:

$$\binom{n}{3} - \sum_{u \in V} \binom{out_u}{2}$$

该内容参考了 eternal 风度的博客[17]。

14.10.6.2 四元环计数

首先对点按照点度升序排序, 然后枚举环上端点 u, 强制 u 为标号最大的点, 然后枚举 u 的相邻点 v, 枚举 v 的相邻点 z, 记录路径 $u \to v \to z$ 的条数 cnt_z, 答案加上 cnt_z, 然后 $++cnt_z$。时间复杂度仍为 $O(m\sqrt{m})$

该方法出自 FWC2019D5T2 的题解。

[15] Lindström–Gessel–Viennot lemma - Wikipedia https://en.wikipedia.org/wiki/Lindstr%C3%B6m%E2%80%93Gessel%E2%80%93Viennot_lemma

[16] 不常用的黑科技——「三元环」https://www.cnblogs.com/KingSann/p/9590525.html

[17] 三元环问题总结 http://www.cnblogs.com/cjoierljl/p/9853236.html

Chapter 15

字符串

15.1 Hash 441
15.1.1 BKDRHash 441
15.1.2 混合 Hash 441
15.1.3 子串 Hash 441
15.2 Trie 字典树 442
15.3 AC 自动机 442
15.3.1 构造 442
15.3.2 查询 443
15.3.3 fail 树 444
15.4 Huffman 编码 444
15.4.1 常规 Huffman 444
15.4.2 k 叉 Huffman 444
15.4.3 效率优化 445
15.4.4 限长 Huffman 445
15.4.5 编码字符代价不等的 Huffman 447
15.5 KMP 算法 448
15.6 Manacher 算法 448
15.7 回文自动机 450
15.7.1 构造 450
15.7.2 应用 451
15.8 后缀树 452
15.8.1 构造 452
15.8.2 广义后缀树 452
15.8.3 应用 452
15.9 后缀数组 453
15.9.1 倍增构造 453
15.9.2 最长公共前缀 454
15.9.3 应用 455
15.10后缀仙人掌 457
15.10.1 概述 457
15.10.2 构造 457
15.10.3 应用 457

15.11后缀自动机 458
15.11.1 描述 458
15.11.2 构造 458
15.11.3 Parent 树的应用 460
15.11.4 线性构造后缀数组 464
15.11.5 SAM with LCT 466
15.11.6 广义 SAM 466
15.11.7 序列自动机 467
15.12算术表达式解析 468
15.13Z Algorithm 470
15.13.1 求解 470
15.13.2 应用 471
15.14卷积法解决字符串匹配问题 471
15.14.1 回文子序列 471
15.14.2 带通配符匹配 475
15.14.3 大字符集处理 475
15.14.4 广义模式匹配 477
15.15最小表示法 477

15.1 Hash

15.1.1 BKDRHash

BKDRHash 在一般情况下表现良好, 速度快, Hash 质量高。代码如下:

```
0 typedef unsigned long long HashT;
  HashT BKDRHash(const char*  str) {
      HashT res=0;
      while(*str) {
          res=res*131+*str;
          ++str;
      }
      return res;
  }
```

15.1.2 混合 Hash

若要快速求得子串 Hash, 预处理前缀 Hash 值, 查询时化到同一幂次再差分。

若要判断两个串是否为同一子串或对称子串, 可以同时预处理后缀 Hash 值, 然后将正向 Hash 值与逆向 Hash 值相乘得到新的 Hash 值。

若要忽略子串的某一连续区间, 可以将其前缀的前缀 Hash 与后缀的后缀 Hash 加权混合([CTSC2014] 企鹅 QQ)。

15.1.3 子串 Hash

有时会遇到与不重复连续子串相关的问题, 可以 $O(n \lg n)$ 枚举(调和级数)。一般来说二分长度 +Hash 是个不错的思路。

15.2 Trie 字典树

Trie 字典树利用了字符串的公共前缀信息, 一般用作搭配 AC 自动机或者实现可持久化 01Trie 回答 xor 最值问题。

Trie 树上每个节点对应一个字符串的某个前缀, 节点的 LCA 为这两个节点所代表字符串的最大公共前缀。

代码如下:

```
struct Node{
    int nxt[26],cnt;
} T[tsiz];
void insert(const char* str) {
    static int icnt=0;
    int p=0;
    while(*str) {
        int& id=T[p].nxt[*str-'a']
        if(id==0)
            id=++icnt;
        p=id;
        ++str;
    }
    ++T[p].cnt;
}
```

有时 Trie 的空间需求很大, 此时需要考虑压缩 Trie 的空间, 尤其是 01Trie。

以 01Trie 的值 $\leq 2^{30}$, 串数 $n \leq 10^6$, 单个节点 16Byte 为例:

保守估计, 开大小 nL 的 Trie:457MB。

注意到最坏情况下是 2^{19} 层被填满,剩下一堆长链,此时数组大小为 $2^{20}+11n$:183MB。

如果剩余链的答案很好统计, 那么干脆将无分支的长链压缩为一点, 该标记往往不需要多余空间。若遇到新分支才把该剩余链向下推。此时数组大小为 $2^{20}+n$:31MB。这个方法需要更多的代码, 其实现要求精细。

由此可见, Trie 的空间优化潜力很大, 但需要冒不小的风险(比如惰性扩展的代码写挂了)。

15.3 AC 自动机

AC 自动机基于 Trie 树实现, 原理类似于 KMP 算法, 即在 Trie 树上匹配字符串, 失配时根据 fail 指针跳到下一匹配位置。

15.3.1 构造

首先建出 Trie, 然后对 Trie 按 BFS 序确定 fail 指针, fail 指针指向该节点在 Trie 中的最长非平凡后缀(根据 Trie 树的性质, 这个后缀也是某个模式串的前缀), BFS 序遍历可以保证其长度最长。

为了简化找 fail 指针的代码, 可以设一个虚拟节点 0, root 编号为 1, 令 root 的 fail 为 0, 0 的所有儿子全设为 root, 这样就可以避免判断是否到达 root, 同时令找不到后缀的节点的 fail 为 root。

```
const int root=1;
int q[size];
```

```
void cook() {
    for(int i=0;i<26;++i)
        T[0].nxt[i]=root;
    T[root].fail=0;
    int b=0,e=1;
    q[b]=root;
    while(b!=e) {
        int u=q[b++];
        for(int i=0;i<26;++i) {
            int v=T[u].nxt[i];
            if(v) {
                int p=T[u].fail;
                while(!T[p].nxt[i])
                    p=T[p].fail;
                T[v].fail=T[p].nxt[i];
                q[e++]=v;
            }
        }
    }
}
```

还有一种更加简洁快速的 cook 过程。让无儿子 v 的节点 u 继承其 fail 的对应儿子，省去了在 Trie 上不断跳 fail 指针的时间。这种 cook 方式不会影响 fail 树的建立。

```
int q[size];
void cook() {
    int b = 0, e = 0;
    for(int i = 0; i < 26; ++i)
        if(T[0].nxt[i])
            q[e++] = T[0].nxt[i];
    while(b != e) {
        int u = q[b++];
        for(int i = 0; i < 26; ++i) {
            int& v = T[u].nxt[i];
            if(v) {
                T[v].fail = T[T[u].fail].nxt[i];
                q[e++] = v;
            } else
                v = T[T[u].fail].nxt[i];
        }
    }
}
```

15.3.2 查询

AC 自动机的经典功能是多模匹配。将主串的字符按顺序在自动机上跳，失配就走 fail 指针，对于每一次匹配后的状态，不断跳 fail 到根查找是否存在节点为某个模式串。

```
void match(const char* str) {
    int p=0;
    while(*str) {
```

```
        int c=*str-'a';
        //注意如果使用继承fail优化就不需要跳fail了
        while(p && !T[p].nxt[c])
            p=T[p].fail;
        p=T[p].nxt[c];
        //查找匹配的模式串
        int cp=p;
        while(cp) {
            if(T[cp].cnt)
                //处理匹配字符串
            cp=T[cp].fail;
        }
        ++str;
    }
}
```

继承方式不影响查询的正确性(继承 fail 节点的儿子相当于跳到 fail 上), 而且同样避免了跳 fail。

15.3.3 fail 树

容易发现 fail 指针所指的节点所代表的字符串是自己的最长非平凡后缀。从 fail 向自己连边, 可以得到一棵 fail 树。将代表某个模式串的节点称为终结点, 那么 fail 树上某个终结点是它子树内的终结点的后缀。

例题 NOI2011 阿狸的打字机 对于多次询问求第 x 个字符串在第 y 个字符串中出现次数的问题, 模拟字符串 y 在 AC 自动机上跑的过程, 查询匹配时跳 fail 扫一遍相当于在 fail 树上将自己的权值传给父亲。因此先建出 fail 树, 按 DFS 序平铺为序列, 让 y 在 AC 自动机上跑, 记录节点被经过的次数, 最后求询问中所有 x 的子树权值和, 使用树状数组很容易实现。

15.4 Huffman 编码

15.4.1 常规 Huffman

计算步骤如下:

1. 将每个字符按照(节点编号, 频率)插入构造最小堆。
2. 弹出频率最低的两个节点, 将这两个节点挂在新节点下, 新节点的频率为这两个节点频率之和, 将新节点插入堆中。
3. 重复步骤 2 直至堆中只剩 1 个节点为止。

15.4.2 k 叉 Huffman

例题 NOI2015 荷马史诗

对于 2 叉 Huffman, 可以保证最后堆中只剩 1 个节点, 但是对于 k 叉 Huffman 来说则不一定, 所以需要 "补零"。考虑 k 叉 Huffman 每次操作都会减少 $k-1$ 个节点, 而且最后剩下 1 个节点, 因此需要补齐 $(k-1-((n-1) \bmod (k-1))) \bmod (k-1)$ 个频率为 0 的空节点。

15.4.3 效率优化

与 NOIP2016 蚯蚓类似, 新插入节点的频率单调非减。因此可以对原字符进行排序, 然后维护两个单调队列, 取两个队列队首最小的一个弹出。

15.4.4 限长 Huffman

论文[1][7] 给出的 Package-Merge Algorithm 用来解决限长 Huffman 问题, 时间复杂度 $O(nL)$。

15.4.4.1 Coin collector's problem

一个硬币收藏家收藏有许多硬币, 每个硬币都有它的面值和收藏价值, 现在硬币收藏家需要选择一个硬币子集, 使得这个子集的面值总和 $= X, X \in \mathbb{N}$, 同时收藏价值总和最小。其中面值取值为 $2^{-k}, 0 \le k \le m$。

15.4.4.2 Package-Merge Algorithm

1. 将硬币按照面额分成多个集合, 集合内按照收藏价值排序;
2. 选取最小面额的集合内的硬币按照顺序两两合并$(1+2, 3+4, \cdots)$后插入 2 倍面额的硬币集合, 若剩余一个硬币则将其丢弃;
3. 不断合并直至合并到面值为 1 的集合, 此时在该集合内贪心选取硬币。

注意插入到下一面值集合的收藏价值是单调非减的, 维护一个指针以支持线性插入。除去排序的复杂度, 该算法复杂度为 $O(n)$。

如果 X 不为最大面值的倍数, 将 X 二进制拆分, 算到某个为 1 的位对应的面值集合时, 选取集合内最小值加入方案并将其从集合内删除, 继续合并。

15.4.4.3 规约到限长 Huffman 问题

设限长为 L, 将面值设为编码后的代价, 即 $2^{-1}, 2^{-2}, \cdots, 2^{-L}$, 设收藏价值为每种字符的频率, 对每个面值都加入所有字符的收藏价值(预先排序), 运行该算法, 最后选取前 $n-1$ 个硬币。使用递归分治可将空间复杂度降为 $O(n)$。

接下来构造标准 Huffman 编码: 统计每个字符被选择的硬币数, 作为其编码的长度(保证长度不超过 L), 计算每个长度的字符数, 然后给当前长度的字符分配位置(即计算偏移), 剩余位置给下一长度的字符使用。

代码如下:

```
0 #include <algorithm>
  #include <cstdio>
  #include <list>
  #include <memory>
  struct CharInfo {
      int id, cnt, len, code;
  } info[10];
  int lenCount[10], off[10];
  class Node;
  using SharedNode = std::shared_ptr<Node>;
```

[1] A fast algorithm for optimal length-limited Huffman codes
http://www.cse.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/p464-larmore.pdf

```
using IterT = std::list<SharedNode>::const_iterator;
template <typename... T>
SharedNode makeSharedNode(T&&... args) {
    return std::make_shared<Node>(
        std::forward<T>(args)...);
}
class Node {
private:
    int val;
    SharedNode ls, rs;

public:
    Node(int val) : val(val) {}
    Node(SharedNode ls, SharedNode rs)
        : val(ls->getWeight() + rs->getWeight()),
          ls(ls), rs(rs) {}
    int getWeight() const {
        return ls ? val : info[val].cnt;
    }
    void count() {
        if(ls) {
            ls->count();
            rs->count();
        } else
            ++info[val].len;
    }
};
int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    if(n > (1 << m))
        return 1;
    for(int i = 0; i < n; ++i) {
        info[i].id = i;
        scanf("%d", &info[i].cnt);
    }
    std::sort(info, info + n, [](const CharInfo& lhs,
                                 const CharInfo& rhs) {
        return lhs.cnt < rhs.cnt;
    });
    std::list<SharedNode> list, stdlist;
    for(int i = 0; i < n; ++i)
        stdlist.emplace_back(makeSharedNode(i));
    list = stdlist;
    for(int i = m; i >= 2; --i) {
        std::list<SharedNode> clist = stdlist;
        std::list<SharedNode> glist;
        IterT it = list.cbegin();
        while(true) {
            IterT a = it++;
```

```
            IterT b = it++;
            if(a != list.cend() && b != list.cend()) {
                glist.emplace_back(
                    makeSharedNode(*a, *b));
            } else
                break;
        }
        glist.merge(clist, [](SharedNode lhs,
                              SharedNode rhs) {
            return lhs->getWeight() < rhs->getWeight();
        });
        glist.swap(list);
    }
    IterT it = list.begin();
    int sum = 2 * (n - 1);
    for(int i = 0; i < sum; ++i, ++it)
        (*it)->count();
    for(int i = 0; i < n; ++i)
        ++lenCount[info[i].len];
    for(int i = 1; i < m; ++i)
        off[i + 1] = (off[i] + lenCount[i]) << 1;
    for(int i = 0; i < n; ++i) {
        int len = info[i].len;
        info[i].code = off[len] + (--lenCount[len]);
    }
    std::sort(info, info + n, [](const CharInfo& lhs,
                                 const CharInfo& rhs) {
        return lhs.id < rhs.id;
    });
    for(int i = 0; i < n; ++i) {
        printf("id=%d code=", i);
        for(int j = info[i].len - 1; j >= 0; --j)
            putchar('0' + ((info[i].code >> j) & 1));
        putchar('\n');
    }
    return 0;
}
```

该算法参考了 Wikipedia-EN[2]和 pymqq 的博客[3]。

15.4.5 编码字符代价不等的 Huffman

现实应用是摩尔斯电码的点和短横线的发送时间不同。

留坑待补, 参见 [6]。

[2]Package-merge algorithm - Wikipedia https://en.wikipedia.org/wiki/Package-merge_algorithm

[3]基于二叉树和双向链表实现限制长度的最优 Huffman 编码 https://blog.csdn.net/pymqq/article/details/32084763

15.5 KMP 算法

KMP 算法通过预处理 nxt 数组来避免重复匹配, 定义 $nxt[i]$ 为长度为 i 的模式串前缀的最长非平凡后缀(同时也是模式串的前缀)长度。

以下字符串下标从 0 开始, nxt 数组下标从 1 开始。

15.5.0.1 预处理

可以利用之前的预处理信息跳 nxt 来找到最长后缀。

```
int nxt[size];
void cook(const char* P) {
    int p=0;
    nxt[1]=0;
    for(int i=1;P[i];++i) {
        while(p && P[p]!=P[i])
            p=nxt[p];
        if(P[p]==P[i])
            ++p;
        nxt[i+1]=p;
    }
}
```

15.5.0.2 匹配

匹配和预处理的过程十分相似。

```
void match(const char* str,int len) {
    int p=0;
    for(int i=0;str[i];++i) {
        while(p && P[p]!=str[i])
            p=nxt[p];
        if(P[p]==str[i])
            ++p;
        if(p==len)
            //match str[i-len+1...i]
    }
}
```

有时 nxt 数组会被用来辅助 dp 转移, 构造出 dp 转移方程后使用矩阵快速幂加速。

ExKMP 已被更好理解的 Z Algorithm 取代, 故不再补充该内容。Z Algorithm 参见第 15.13节。

15.6 Manacher 算法

Manacher 算法用来求解最长回文串问题。主要思想是利用之前的计算结果来加速回文串的计算, 从而达到线性时间复杂度。

算法步骤如下:

1. 为了统一奇回文串和偶回文串, 向每对相邻字符间插入一个未出现过的字符, 接下来算法仅讨论奇回文串;

2. 维护当前访问到的最右位置 $maxr$ 和最右位置所对应的字符串中心 pos, 以及以每个位置为中心向右扩展长度 $RL[i]$(从中心开始数);

3. 对于每一个位置:

   (a)
   - 若当前位置 $i \geq maxr$, 设 $RL[i] = 1$;
   - 否则令 $RL[i] = min(RL[pos - (i - pos)], maxr - i + 1)$ (因为此时 i 到 $maxr$ 的部分和 i 以 pos 为轴的对称部分对称, 而那个部分已经被处理过了)。

   (b) 不断向两端扩展增大 $RL[i]$;

   (c) 更新 $maxr$ 与 pos。

4. 答案即为 $RL[i] - 1$ 的最大值。

小 trick

- 在字符串开头再加另一个特殊字符, 可以不用越界检查。
- 令 maxr 为当前匹配最右边的位置的右边一位, 避免 +1-1 的麻烦。

这两个 trick 源自小蒟蒻 yyb 的博客[4]。
代码如下:

```
0 char buf[size],str[2*size];
  int RL[2*size];
  int manacher() {
      int cnt=0;
      str[cnt++]='#';
      str[cnt++]='@';
      for(int i=0;buf[i];++i) {
          str[cnt++]=buf[i];
          str[cnt++]='@';
      }
10    int maxr=0,pos=0,ans=0;
      for(int i=1;i<cnt;++i) {
          RL[i]=(maxr>i?std::min(RL[2*pos-i],maxr-i):1);
          while(str[i-RL[i]]==str[i+RL[i]])
              ++RL[i];
          if(i+RL[i]>maxr) {
              maxr=i+RL[i];
              pos=i;
          }
          ans=std::max(ans,RL[i]);
20    }
      return ans-1;
  }
```

[4]【BZOJ3160】万径人踪灭(FFT, Manacher)https://www.cnblogs.com/cjyyb/p/8435460.html

15.7 回文自动机

15.7.1 构造

回文自动机的每一个节点对应一个**本质不同的回文串**。

回文自动机需要维护每个节点所对应的回文串的长度 len, 两端加入字符 c 后的后继节点 $nxt[c]$, 自身的最长后缀回文串所对应的节点 $fail$, 以及自身代表的回文串数(需要最后在 fail 树上上传才是完整的)。还要维护当前已加入自动机的字符 buf, 以及最后一次加入字符后的状态 $last$。

构造 PAM 的复杂度为 $O(n \lg |\Sigma|)$。

15.7.1.1 初始化

首先在空 PAM 中加入两个根:偶数长度的根 0 和奇数长度的根 1。其中节点 0 的 fail 指向 1, len 为 0, 节点 1 的 len 为-1(避免特判)。同时令 $buf[0] = -1$(避免特判)。

15.7.1.2 状态转移

构造 PAM 时, 按顺序向 PAM 加入字符。首先向 buf 加入该字符, 然后在 fail 树上跳最长后缀回文串, 直至找到对称点与自身相同为止。注意如果找不到这样的对称点, 就会到达奇数长度的根, 它的 len 为-1, 其对称点就是自己, 所以迭代必定会结束。

接下来查看是否有该节点的后继节点, 如果没有就新建一个节点, len 比父亲多 2, fail 指针重新在父亲的 fail 链上找。然后把该节点挂在父亲下。

最后重置 last,++T[last].cnt 结束加入。

15.7.1.3 后处理

注意自身的 fail 肯定是自己的子回文串, 最后做一次后处理, 按照节点编号的逆序往上累加 cnt。

15.7.1.4 代码

```
#include <algorithm>
#include <cstdio>
const int size = 300005;
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
struct PAM {
    struct Node {
        int len, nxt[26], fail, cnt;
    } T[size];
    char buf[size];
    int last, siz, n;
    PAM() {
        T[0].len = 0;
        T[0].fail = 1;
        T[1].len = -1;
        last = 0;
        siz = 1;
        n = 0;
        buf[0] = -1;
```

```
    }
    int getFail(int p, int c) const {
        while(buf[n - 1 - T[p].len] != c)
            p = T[p].fail;
        return p;
    }
    void extend(int c) {
        buf[++n] = c;
        int p = getFail(last, c);
        if(!T[p].nxt[c]) {
            int u = ++siz;
            T[u].len = T[p].len + 2;
            T[u].fail =
                T[getFail(T[p].fail, c)].nxt[c];
            T[p].nxt[c] = u;
        }
        last = T[p].nxt[c];
        ++T[last].cnt;
    }
    Int64 solve() {
        Int64 ans = 0;
        for(int i = siz; i >= 1; --i) {
            T[T[i].fail].cnt += T[i].cnt;
            ans = std::max(ans, asInt64(T[i].cnt) *
                                T[i].len);
        }
        return ans;
    }
} pam;
int main() {
    int c;
    do
        c = getchar();
    while(c < 'a' || c > 'z');
    while('a' <= c && c <= 'z') {
        pam.extend(c - 'a');
        c = getchar();
    }
    printf("%lld\n", pam.solve());
    return 0;
}
```

15.7.2 应用

15.7.2.1 统计串 S 的前缀本质不同的回文串个数

extend 该前缀的所有字符后，PAM 的 $siz - 1$ 就是本质不同的回文串个数。

15.7.2.2 统计串 S 的每个本质不同的回文串的出现次数

由于 PAM 中每个节点代表一个回文串, 后处理后每一个节点的 cnt 就是它的出现次数。

15.7.2.3 统计串 S 的回文串的个数

答案即为后处理后的 cnt 之和。

15.7.2.4 统计以下标 i 结尾的回文串个数

对每个节点记录其在 fail 树上的深度, fail 链上的节点所代表的回文串都是自己的后缀回文串, 所以答案即为加入第 i 字符后 last 节点的深度。

以上内容参考了 poursoul 的博客[5]。

15.8 后缀树

后缀树是由串 S 的所有后缀组成的压缩 Trie。

为了避免压缩过程隐藏了某个后缀, 插入每个后缀后再插入一个不出现的字符。

15.8.1 构造

常规的构造方法是 $O(n^2)$ 的, 一般采用 Ukkonen 算法在线性时间内完成构造。 留坑待补(还是去学后缀仙人掌吧)。

15.8.2 广义后缀树

广义后缀树在插入不同的串时用的结尾字符不同, 以区分不同的字符串。

查询最长公共子串时要找到最深的拥有 2 种结束标记的节点。

15.8.3 应用

后缀树满足如下性质:

- 每个节点代表一个子串。

  查询串 P 在串 S 的出现次数可以按照 Trie 的方法匹配, 然后统计其所在节点的子树中的叶子节点个数。

- 每个非叶节点至少有两个儿子。

  统计串 S 的最长重复子串要找到其最深非叶子节点。

以上内容参考了 v_JULY_v 的博客[6]。

[5]Palindromic Tree——回文树【处理一类回文串问题的强力工具】https://blog.csdn.net/u013368721/article/details/42100363

[6]从 Trie 树 (字典树) 谈到后缀树 (10.28 修订) https://blog.csdn.net/v_july_v/article/details/6897097

15.9 后缀数组

15.9.1 倍增构造

后缀数组 $SA[i]$ 表示排名为 i 的后缀位置, 相应地可以得到起始位置为 i 的后缀排名 $rk[i]$。

数组 rk 一般使用倍增法 $O(n \lg n)$ 构造, 然后对应地初始化 SA。

构造步骤如下:

1. 对于每一个字符初始化其排序关键字(即字符);
2. 对关键字进行基数排序, 算出此时字符串 $[i \dots i + 2^k - 1]$ 的排名;
3. 若所有排名都不相同, 直接跳出; 否则将 $rk[i]$ 与 $rk[i + 2^k]$ 作为以 i 为起始位置的字符串的关键字, 重复步骤 2;
4. 根据 rk 数组初始化 SA 数组。

代码如下:

```
#include <algorithm>
#include <cctype>
#include <cstdio>
const int size = 1000005;
typedef long long Key;
Key makeKey(Key id, Key a, Key b) {
    return (id << 20 | a) << 20 | b;
}
template <int off>
int getKey(Key k) {
    return (k >> off) & 0xfffff;
}
int cnt[size];
template <Key* A, Key* B, int off>
void radixSortImpl(int n) {
    int maxw = 0;
    for(int i = 1; i <= n; ++i) {
        int cw = getKey<off>(A[i]);
        maxw = std::max(maxw, cw);
        ++cnt[cw];
    }
    for(int i = 1; i <= maxw; ++i)
        cnt[i] += cnt[i - 1];
    for(int i = n; i >= 1; --i)
        B[cnt[getKey<off>(A[i])]--] = A[i];
    for(int i = 0; i <= maxw; ++i)
        cnt[i] = 0;
}
Key A[size], B[size];
int rk[size];
bool radixSort(int n) {
    radixSortImpl<A, B, 0>(n);
```

```
    radixSortImpl<B, A, 20>(n);
    int crk = 0;
    Key lw = 0;
    for(int i = 1; i <= n; ++i) {
        int id = A[i] >> 40;
        Key cw = A[i] & 0xffffffffff;
        if(lw != cw) {
            ++crk;
            lw = cw;
        }
        rk[id] = crk;
    }
    return crk == n;
}
int sa[size];
int main() {
    int c, n = 0;
    do
        c = getchar();
    while(!isgraph(c));
    while(isgraph(c)) {
        ++n;
        A[n] = makeKey(n, c, 0);
        c = getchar();
    }
    int off = 1;
    while(!radixSort(n)) {
        for(int i = 1; i <= n; ++i) {
            int sec = i + off <= n ? rk[i + off] : 0;
            A[i] = makeKey(i, rk[i], sec);
        }
        off <<= 1;
    }
    for(int i = 1; i <= n; ++i)
        sa[rk[i]] = i;
    for(int i = 1; i <= n; ++i)
        printf("%d ", sa[i]);
    return 0;
}
```

15.9.2 最长公共前缀

定义 $LCP(i,j)$ 为第 i 个后缀与第 j 个后缀(指排名而不是位置)的最长公共前缀。显然 LCP 函数有两个性质:

- $LCP(i,j) = LCP(j,i)$
- $LCP(i,i) = n - SA[i] + 1$

所以只要考虑 $LCP(i,j), i < j$ 的情况。

定理 15.1 (LCP Theorem)

$$LCP(i,j) = min\{LCP(k-1,k)\}, i < k \leq j$$

要证明该定理可先证明下列引理:

引理 15.2 (LCP Lemma)

$$LCP(i,j) = min(LCP(i,k), LCP(k,j)), i < k < j$$

证明 首先根据传递性显然有 $LCP(i,j) \geq min(LCP(i,k), LCP(k,j))$。

其次考虑到 i,j,k 是后缀的排名且 $i < k < j$, 有 $LCP(i,j) \leq LCP(i,k)$ 且 $LCP(i,j) \leq LCP(k,j)$, 即 $LCP(i,j) \leq min(LCP(i,k), LCP(k,j))$。该引理得证。

因此设 $height[i] = LCP(i-1,i), height[1] = 0$, 求出 height 数组后构建 ST 表 $O(1)$ 询问 LCP。

按照原串顺序预处理 $height$ 数组。考虑在原串上相邻后缀的关系, 显然两个后缀右移 1 位的 $LCP' \geq LCP - 1$, 用此性质快速转移后继续向后暴力匹配。

```
0 void cook(int n) {
      int h=0;
      for(int i=1;i<=n;++i) {
          if(rk[i]==1)h=0;
          else {
              int k=SA[rk[i]-1];
              if(h)--h;
              while(buf[i+h]==buf[k+h])
                  ++h;
          }
10        height[rk[i]]=h;
      }
  }
```

以上内容参考了 Angel_Kitty 的博客[7]。

15.9.3 应用

15.9.3.1 一般思路

- 二分 LCP 长度, 对 height 数组进行分组。
- 若遇到多串则将其用未出现字符连接后求后缀数组。
- 对于回文串/翻转系列问题则将其与反串用特殊字符相连后求后缀数组。
- 连续重复子串问题使用错位匹配解决。
- 「TJOI / HEOI2016」字符串:若 LCP 的一个端点固定, 可以从它开始在 height 数组上左右暴力遍历, 利用 min 单调性剪枝。
- 枚举串长 l, 每 l 个位置分一段, 长度为 l 的串肯定会跨过一个关键点, 时间复杂度 $O(n \lg n)$。

[7]后缀数组 (一堆干货) - Angel_Kitty https://www.cnblogs.com/ECJTUACM-873284962/p/6618870.html

15.9.3.2 可重叠最长重复子串

该子串一定是某两个后缀的 LCP, 而 LCP 在 height 数组中取最小值, 因此答案为 height 最大值。

15.9.3.3 不可重叠最长重复子串

二分 LCP 长度 k, 按 k 对 height 数组进行划分, 满足每块内的 height 值 $\geq k$, 判断是否存在块内 $SA[i]$ 的极差 $\geq k$(此时子串不重叠)。

15.9.3.4 可重叠 k 次最长重复子串

二分 LCP 长度, 对其分组, 询问是否存在大小 $\geq k$ 的块。

15.9.3.5 本质不同的子串个数

考虑按照后缀字典序加入每个后缀的前缀, 每个后缀贡献了 $n - SA[i] + 1$ 个前缀, 去掉重复的 $height[i]$ 个重复前缀。答案为每个后缀的贡献之和。可以把这两部分分开考虑, 答案为子串数-height 数组和。

15.9.3.6 最长回文子串

将串与反串用未出现字符连接后求后缀数组, 按照长度奇偶分类讨论 (或者类似于 Manacher 算法处理原串), 枚举对称中心, 求以其为首的后缀与以其在反串上的对称位置为首的后缀的 LCP。

15.9.3.7 连续重复子串

已知字符串 S 由某个字符串多次重复得到, 求最大重复次数。

枚举串长 n 的因子 k, 询问字符串 $[1 \dots n-k]$ 与 $[k+1 \dots n]$ 是否相等, 即判断 $LCP(rk[1], rk[k+1]) = n - k$ 是否成立。由于 LCP 的一端是固定的, 没有必要构建 ST 表支持 RMQ, 可以直接 $O(n)$ 扫描处理。

15.9.3.8 重复次数最多的连续重复子串

首先枚举连续重复子串长度 L, 仅考虑重复 2 次以上的情况, 那么整个连续重复串每隔 L 个必相同, 可以枚举起始位置 $L * i, L * (i+1)$ 错位匹配求 LCP。若起点不为 L 的倍数, 尝试计算 LCP 判断左边剩余部分是否相等。

15.9.3.9 最长公共子串

将两个串拼接在一块, 求满足对应后缀起点来自不同字符串的 height 最大值。

15.9.3.10 长度 $\geq k$ 的公共子串数(可以相同)

按 k 对 height 划分, 在块内统计每个后缀之前的来自另一个字符串的后缀与该后缀产生的贡献, 这里可以用单调栈维护以当前位置结尾的 height 后缀最小值之和。分别对两个串的后缀扫一遍累积贡献即为答案。

15.9.3.11 出现于不少于 k 个字符串的最长子串

二分长度对 height 分组, 然后检查每组内是否存在来自 k 个字符串的后缀。

15.9.3.12 在每个字符串重复但不重叠的最长子串

二分长度对 height 分组, 对每个字符串检查重复且不重叠的条件。

15.9.3.13 出现或翻转后出现在每个字符串中的最长子串

将每个串的原串 + 反串连起来, 二分长度对 height 分组, 判断是否满足存在来自所有字符串的后缀。

以上内容参考了罗穗骞的论文《后缀数组——处理字符串的有力工具》。

15.10 后缀仙人掌

15.10.1 概述

后缀仙人掌将后缀 Trie 的每个非叶子结点与它的一个儿子合并, 形成 "树枝"。每个树枝表示的是从根节点到自身叶子节点的后缀。树枝的深度为其顶端节点的父节点的深度, 根树枝的深度为 0。记后缀排名为 s 的后缀为 s, 深度为 $depth(s)$。一个树枝的父亲树枝为包含其顶端节点左兄弟的树枝。

后缀仙人掌有如下性质:

性质 15.3 $depth(s) = LCP(SA[s], SA[s-1])$

显然 $depth(s)$ 是它和它前一个后缀的最小公共祖先的深度。$depth(s)$ 其实就是 $height[s]$。

性质 15.4 树枝 $r(r > 1)$ 的父亲树枝是满足 $depth(s) \leq depth(r), s < r$ 的编号最大的树枝 s。

15.10.2 构造

根据 $depth(s)$ 与 $height[s]$ 的联系, 首先预处理后缀数组与 $height$ 数组。

然后按照字典序从小到大考虑, 维护一个单调栈, 栈内存储可能的父亲树枝编号, 找到需要的父亲树枝 k 后将边 $(k, depth(s)) = s$ 插入 HashTable。

代码如下:

```
int st[size];
void buildSC(int n) {
    int top=1;
    st[top]=1;
    for(int i=2;i<=n;++i) {
        while(height[st[top]]>height[i])
            --top;
        addEdge(st[top],height[i],i);
        st[++top]=i;
    }
}
```

15.10.3 应用

多次询问串 T 的前缀为串 S 的子串的最大长度。

维护当前所在树枝 id, 与匹配长度 d。每字符匹配时尽可能沿树枝走, 走不到就跳到子树枝, 若没有子树枝则返回。

```
int match(const char* str,int n) {
    int id=1,d=0,i;
    for(i=0;str[i];++i) {
        char c=str[i];
        bool flag=false;
        while(sa[id]+d>n || buf[sa[id]+d]!=c) {
            int nxt=find(id,d);
            if(!nxt) {
                flag=true;
                break;
            }
            id=nxt;
        }
        if(flag)
            break;
        ++d;
    }
    return i;
}
```

上述内容参考了 WC2014 营员交流课件《Suffix Cactus》。

15.11 后缀自动机

15.11.1 描述

后缀自动机(SAM)可用来识别母串的后缀, 使用最简状态表示。SAM 是一个 DAG, 任何从初始状态出发的路径对应母串的某个子串。与其他自动机不同的是, SAM 中一个状态对应多个子串。

15.11.2 构造

假设现在已经构造出了串 S 的 SAM, 考虑如何构造串 Sx 的 SAM 以识别新的子串。如果能够快速完成这个任务, 就可以逐位将字符插入 SAM 以构造出整个串的 SAM。

为了加强理解, 这里不直接按照最终的 SAM 描述, 而是需要什么元素加入什么元素。~~兵来将挡~~, ~~水来土掩~~。记 $SAM(P)$ 为串 P 的 SAM, $S_P(i)$ 表示 P 的第 i 个后缀。

首先用根节点表示空串, 记为 R。

考虑由 $SAM(P)$ 构造出 $SAM(Px)$, 添加字符 x 使得当前串新增后缀 $x, S_P(1)x, \cdots, S_P(|P|)x$。那么可以考虑新增一个节点, 将 $SAM(P)$ 表示后缀的节点向它连一条 x 转移边。注意到上一次增加的节点(记为 $last(P)$)表示了 $S_P(1), \cdots, S_P(|P|)$, 需要连边的只有与 R 与 $last(P)$。此时每个节点需要存储转移边 $nxt[|\Sigma|]$。

注意到每个节点的相同字母转移边只能有一条, 而上述方法中节点 R 一直在连边, 当已有转移边时就无法转移了。考虑向串 ab 后加字符 a, R 已有字符 a 的转移边。注意到若不连该边, 原有的连边可以识别后缀 a, 不过下次转移时表示后缀的节点不再是 R 与 $last(P)$, 而是 $R, last(P)$ 与表示 a 的节点。可以考虑用一条链将它们串起来, 从 $last(Px)$ 开始按照最长串长度连向上一个表示后缀的节点, 此时每个节点需要存储该节点的 $link$。

如何程序化地描述向新点 id 连转移边的过程呢? 那就是从 $last(P)$ 开始, 若该节点没有转移边则将其连向新点, 然后根据 $link$ 指针跳到下一个表示后缀的节点。若跳到 R 处也没有转移边, 则设置新点的 $link = R$, 更新 $last(Px) = id$, 退出程序。否则说明 $SAM(P)$

已经能够识别该后缀, 只需设置 id 的 $link$。记当前有转移边的节点为 p, 转移边指向 q。注意到节点 q 不一定表示 p 表示的后缀 + 字符 x, 需要进行判断。

考虑按照最长串长度排列的 $link$ 链, 既然链上所有节点对应了所有后缀, 且链上节点代表的最长串长度从 $last(P)$ 递减, 那么每一个节点代表的后缀长度是一个连续区间, 且区间之间无缝。此外节点还有一个性质: 节点编号 + 串长度唯一对应一个子串。证明: 若节点编号与串长度相同, 而代表的子串不同, 则说明这两条不重叠的路径等长, 将这两条路径从当前节点延伸到某个叶子, 对应了两个长度相等且不相同的后缀, 与事实矛盾。

有了这两个性质, 可以推出一个结论: 若节点 q 的最长串长度 len_q 恰好等于节点 p 的最长串长度 $len_p + 1$, 则节点 q 及其在 $link$ 链上往 R 的方向的元素代表了剩余的新后缀 $(S_P(0 \cdots len_p) + x \to S_{Px}(1 \cdots len_q)$, 还有链头 R 代表空串, 而长度 $\geq len_p + 2$ 的后缀已经在连转移边后可被 id 识别)。此时每个节点需要存储最长串长度 len。若 $len_p + 1 = len_q$, 则令 $link_{id} = q$。否则考虑分割出我们需要的后缀, 即把节点 q 代表的子串切割为两部分, 其中一部分的最长串长度为 $len_p + 1$。可以新建一个节点 cq, 由于切出的这一个子串也能转移到新的节点, 所以需要拷贝 q 的转移边。由于 $len_{cq} < len_q$, 需要修改 $link$ 链为 $link_q \leftarrow cq \leftarrow q$, 最后将 id 的 $link$ 指向 cq。注意分割后原来转移边指向 q 的节点 p 及它在 $link$ 链上的节点都应该改指向 cq, 因为原本它们指向的就是这一部分子串(只是因为未分离前合并到指向 q)。无论是修改转移边到 id 还是 cp, 都是从 $last(P)$ 或 p 开始的连续子链, 因为之前修改转移边的操作影响的都是连续子链。

SAM 的点数不超过 2n-2, 边数不超过 3n-3 (转移边 +Parent 树边), 构造复杂度为 $O(n|\Sigma|)$, 证明参见文末引用。

代码如下:

```
struct SAM {
    struct State {
        int nxt[26],link,len;
    } S[size*2];
    int last,siz;
    SAM():last(1),siz(1) {}
    void extend(int c) {
        int id=++siz;
        S[id].len=S[last].len+1;
        int p=last;
        while(p && !S[p].nxt[c]) {
            S[p].nxt[c]=id;
            p=S[p].link;
        }
        if(p) {
            int q=S[p].nxt[c];
            if(S[p].len+1==S[q].len)
                S[id].link=q;
            else {
                int cq=++siz;
                S[cq]=S[q];
                S[cq].len=S[p].len+1;
                while(p && S[p].nxt[c]==q) {
                    S[p].nxt[c]=cq;
                    p=S[p].link;
                }
                S[q].link=S[id].link=cq;
            }
```

```
            }
            else S[id].link=1;
            last=id;
        }
    }
```

15.11.3 Parent 树的应用

将 *link* 链并在一起, 就得到了一棵树, 称为 Parent 树。

15.11.3.1 Right 集合

节点 u 表示的所有子串的结束位置集合称为节点 u 的 Right 集合。

15.11.3.2 Parent 树性质

定理 15.5 父节点的 *Right* 集合是儿子 *Right* 集合的并。

证明:父节点表示的子串同时也是儿子表示的子串的后缀。

定理 15.6 不在同一条 *link* 链上的节点 *Right* 集合不相交。

证明留坑待补。

这个性质保证了可持久化线段树合并的复杂度。

Parent 树自底向上 Right 集合逐渐变大, 匹配子串长度逐渐变小。据此性质可贪心倍增跳到满足条件的最高的祖先后利用该位置的数据查询。

15.11.3.3 子串匹配

注意 Parent 树上的父亲是儿子的后缀, 因此匹配子串时可以在转移边上跑, 失配就跳 Parent 树的 link(等同于 fail 树)。

15.11.3.4 与后缀树的联系

Parent 树是反串的后缀树, 因为父亲是儿子的后缀, 等同于父亲的反串是儿子反串的前缀, 且该树可以识别反串的所有后缀。

15.11.3.5 计数问题

对于每一个状态维护一个 *right* 值表示当前状态的 Right 集合大小。新增状态时该状态贡献了 1, 但注意克隆状态并没有贡献, 所以克隆后令 $cq.right = 0$。最后拓扑排序 dp 在 Parent 树上自底向上更新就可以得到真实 right 值。

状态 s 表示了 $s.len - s.link.len$ 个本质不同的子串, 每种子串有 $s.right$ 个。

优化: 拓扑排序时可以按照 len 进行分层基数排序, 但是广义 SAM 不能使用这种方法。

15.11.3.6 可持久化线段树合并维护 Right 集合

有时需要判定某个终点是否在某个状态的 Right 集合内, 可以在 extend 时给新建状态添加对应的 Right 值, 然后拓扑排序进行线段树合并计算出每个状态真正的 Right 集合。时间复杂度 $O(n \lg n)$。

若该方法用于匹配母串的某个子串的子串,在失配时沿着 Parent 树跳跃,注意要逐位跳,当匹配长度等于父亲的最长后缀时才跳到父亲。例题:NOI2018 你的名字

15.11.3.7 倍增自匹配

例题 「TJOI / HEOI2016」字符串

通过前缀 → 后缀以及二分 LCP 长度将原问题转换为:给定子串 a 与 b, 求长度为 m 的 b 的后缀是否出现在子串 a 中。

由于 a 与 b 在同一个串内, 该问题即为判定 Right 集合中含有 b 的 Right 值且 $r_S \geq m$ 的状态 S 与 a 对应的 Right 区间是否有交。可以在构建 SAM 时保存每个字符对应的 $last$, 这些状态一定含有对应字符的 Right 值。由于 Parent 树上 Right 集合的包含关系, 其祖先也有该 Right 值且 Right 集合更大, r_S 递减, 可以使用倍增计算出极大的 Right 集合所对应的状态(要求有 b 的 Right 值且 $r_S \geq m$), 然后线段树查询该状态的 Right 集合是否与指定区间有交。

代码:

```
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005, maxS = size * 2;
struct SAM {
    struct Node {
        int len, link, nxt[26];
    } T[maxS];
    int last, siz;
    SAM() : last(1), siz(1) {}
    int extend(int ch) {
        int p = last, id = ++siz;
        T[id].len = T[last].len + 1;
        while(p && !T[p].nxt[ch]) {
            T[p].nxt[ch] = id;
            p = T[p].link;
        }
        if(p) {
            int q = T[p].nxt[ch];
            if(T[q].len == T[p].len + 1)
                T[id].link = q;
            else {
                int cq = ++siz;
                T[cq] = T[q];
                T[cq].len = T[p].len + 1;
                while(p && T[p].nxt[ch] == q) {
                    T[p].nxt[ch] = cq;
                    p = T[p].link;
                }
```

```
                T[q].link = T[id].link = cq;
            }
        } else
            T[id].link = 1;
        last = id;
        return id;
    }
} sam;
struct Node {
    int ls, rs;
} T[maxS * 20];
int icnt = 0;
int build(int l, int r, int pos) {
    int id = ++icnt;
    if(l != r) {
        int m = (l + r) >> 1;
        if(pos <= m)
            T[id].ls = build(l, m, pos);
        else
            T[id].rs = build(m + 1, r, pos);
    }
    return id;
}
int merge(int u, int v) {
    if(u && v) {
        int id = ++icnt;
        T[id].ls = merge(T[u].ls, T[v].ls);
        T[id].rs = merge(T[u].rs, T[v].rs);
        return id;
    }
    return u | v;
}
bool find(int l, int r, int id, int nl, int nr) {
    if(id == 0)
        return false;
    if(nl <= l && r <= nr)
        return true;
    int m = (l + r) >> 1;
    if(nl <= m && find(l, m, T[id].ls, nl, nr))
        return true;
    if(m < nr && find(m + 1, r, T[id].rs, nl, nr))
        return true;
    return false;
}
int n, p[20][maxS], pos[size], root[maxS], q[maxS],
    cnt[size];
void build() {
    for(int i = 1; i <= sam.siz; ++i)
        ++cnt[sam.T[i].len];
    for(int i = 1; i <= n; ++i)
```

```
        cnt[i] += cnt[i - 1];
    for(int i = 1; i <= sam.siz; ++i)
        q[cnt[sam.T[i].len]--] = i;
    for(int i = 1; i <= n; ++i)
        root[pos[i]] = build(1, n, i);
    for(int i = sam.siz; i >= 1; --i) {
        int u = q[i], fa = sam.T[u].link;
        root[fa] = merge(root[fa], root[u]);
    }
    for(int i = 1; i <= sam.siz; ++i)
        p[0][i] = sam.T[i].link;
    for(int i = 1; i < 20; ++i)
        for(int j = 1; j <= sam.siz; ++j)
            p[i][j] = p[i - 1][p[i - 1][j]];
}
bool check(int len, int l, int r, int u) {
    for(int i = 19; i >= 0; --i)
        if(sam.T[p[i][u]].len >= len)
            u = p[i][u];
    return find(1, n, root[u], l, r);
}
char buf[size];
int mini(int a, int b) {
    return a < b ? a : b;
}
int main() {
    n = read();
    int m = read();
    for(int i = 1; i <= n; ++i)
        buf[i] = getchar();
    int rev = n + 1;
    for(int i = n; i >= 1; --i)
        pos[rev - i] = sam.extend(buf[i] - 'a');
    build();
    while(m--) {
        int b = rev - read();
        int a = rev - read();
        int d = rev - read();
        int c = rev - read();
        int l = 1, r = mini(b - a, d - c) + 1, ans = 0;
        while(l <= r) {
            int m = (l + r) >> 1;
            if(check(m, a + m - 1, b, pos[d]))
                ans = m, l = m + 1;
            else
                r = m - 1;
        }
        printf("%d\n", ans);
    }
    return 0;
```

}

15.11.4 线性构造后缀数组

首先构造出 SAM, 发现 last 到根的链上的状态分别代表每一个后缀。对这些状态进行标记, 按照字典序 DFS, 维护 DFS 子串的长度 d, 通过遍历顺序得到 sa 数组。

注意对于跑单条链的情况要使用路径压缩优化。

代码如下:

```
// P4051
#include <cstdio>
const int size = 100005, maxS = size * 4, esiz = 95;
struct SAM {
    struct State {
        int len, link, nxt[esiz], jmp, jdis;
        bool mark;
    } st[maxS];
    int siz, last;
    SAM() {
        siz = last = 1;
    }
    void extend(int c) {
        int id = ++siz;
        st[id].len = st[last].len + 1;
        int p = last;
        while(p && !st[p].nxt[c]) {
            st[p].nxt[c] = id;
            p = st[p].link;
        }
        if(p) {
            int q = st[p].nxt[c];
            if(st[p].len + 1 == st[q].len)
                st[id].link = q;
            else {
                int cq = ++siz;
                st[cq] = st[q];
                st[cq].len = st[p].len + 1;
                while(p && st[p].nxt[c] == q) {
                    st[p].nxt[c] = cq;
                    p = st[p].link;
                }
                st[q].link = st[id].link = cq;
            }
        } else
            st[id].link = 1;
        last = id;
    }
    int len() const {
        return st[last].len;
    }
```

```
    void jump(int u) {
        int p, dis = 0;
        for(p = u; st[p].jmp; p = st[p].jmp)
            dis += st[p].jdis;
        if(dis) {
            int res = p;
            for(p = u; st[p].jmp;) {
                int old = st[p].jdis;
                st[p].jdis = dis;
                dis -= old;
                int nxt = st[p].jmp;
                st[p].jmp = res;
                p = nxt;
            }
        }
    }
    int top, *sa;
    void DFS(int u, int d) {
        if(st[u].mark)
            sa[++top] = d;
        for(int i = 0; i < esiz; ++i) {
            int v = st[u].nxt[i];
            if(v) {
                jump(v);
                if(st[v].jmp)
                    DFS(st[v].jmp, d - 1 - st[v].jdis);
                else
                    DFS(v, d - 1);
            }
        }
    }
    void buildSA(int* buf) {
        for(int p = last; p; p = st[p].link)
            st[p].mark = true;
        for(int i = 1; i <= siz; ++i)
            if(!st[i].mark) {
                int c = 0, p;
                for(int j = 0; j < esiz && c <= 1; ++j) {
                    int v = st[i].nxt[j];
                    if(v) {
                        p = v;
                        ++c;
                    }
                }
                if(c == 1)
                    st[i].jmp = p, st[i].jdis = 1;
            }
        sa = buf;
        top = 0;
        DFS(1, len() + 1);
```

```
    }
} sam;
char buf[size];
int SA[2 * size];
int main() {
    gets(buf);
    for(int i = 0; buf[i]; ++i)
        sam.extend(buf[i] - ' ');
    int len = sam.len();
    for(int i = 0; buf[i]; ++i)
        sam.extend(buf[i] - ' ');
    sam.buildSA(SA);
    for(int i = 1; i <= sam.top; ++i)
        if(SA[i] <= len) {
            int p = SA[i] - 2;
            if(p == -1)
                p += len;
            putchar(buf[p]);
        }
    putchar('\n');
    return 0;
}
```

还可以利用 Parent 树的性质得到更优的做法。反串的 SAM 的 Parent 树就是原串的后缀树, 对后缀树按照字典序遍历后就可以得到 rank 与 sa 数组。

这个算法的关键在于如何计算出字典序顺序, 即每个节点的转移边(转移边不会重复, 如果重复就会开新的公共点)。首先沿着转移边 DFS, 仅找 len 比自身大 1 的后继 v 以避免重复遍历。然后将转移字符压入栈中, 设置 $link_v$ 的第 $len_v - len_{link_v}$ 个字符转移为 v, 这恰好是 $link_v$ 到 v 转移边上的首字符。第二次按照字典序贪心 DFS 一遍就可以得到后缀数组。

该方法参考了 z1j1n1 的博客[8]。

15.11.5 SAM with LCT

为了支持在线修改查询, 需要使用 LCT 维护 Parent 树信息(一般与 Right 集合有关)。

注意 LCT 维护的是有根树, link 和 cut 时要特别注意。事实上由于 Right 集合的维护只需要从自身加到根, 因此可以在 link 时提取出从父亲到根的链, 然后做区间加法, 这样就不要 update, 取而代之的是 pushDown 更新标记。

在取用节点子树信息时需要更新标记, 因此要调用 $pushDown$, 不过为了保证复杂度最好还是使用 $splay$。

注意节点 cq 的父亲是 T[cq].link 而不是 p。

该内容参考了 Candy? 的博客[9]。

15.11.6 广义 SAM

有两种构造方法:

[8]使用后缀自动机求后缀数组
https://www.cnblogs.com/zhujiangning/p/7999381.html

[9]BZOJ 2555: SubString [后缀自动机 LCT] https://www.cnblogs.com/candy99/p/6377537.html

- 在线:插入一个字符串之前将 $last$ 重置,时间复杂度为 O(Trie 大小 * 字符集大小 + 叶子状态深度和)。
- 离线:先建出 Trie,BFS 插入,插入时把父亲在 SAM 上的编号当做 $last$,时间复杂度为 O(Trie 大小 * 字符集大小)。

15.11.6.1 统计状态对应的模板串数

为每个节点记录最后匹配的模板串编号,每次 extend 后从 $last$ 开始沿着 parent 树暴力上跳,将这些节点的 $count$ 值 $+1$ 并标记,直到遇到被同模板串标记过节点的为止。设状态总数为 S,最坏时间复杂度 $O(S\sqrt{2S})$,一般情况下不会达到最坏情况,有时跑得比 lg 做法还快。在想不出更优做法时,这是一个简单有效的方法。

以上内容参考了 WC2012 陈立杰的讲课课件《后缀自动机 Suffix Automaton》与 Candy?[10]、dwjshift[11]、Mangoyang[12]的博客。Menci 的博客写得更详细,一些性质的证明请移步 https://oi.men.ci/suffix-automaton-notes/。

15.11.7 序列自动机

类比后缀自动机,序列自动机上的每条路径对应一个子序列。

15.11.7.1 构造

序列自动机的构造比较简单,即预处理 $nxt[i][j]$ 表示位置 i 后的第一个字符 j 出现的位置。存在一个简单的 $O(n|\alpha|)$DP:

```
0 for(int i=n;i>=1;--i) {
      for(int j=0;j<26;++j)
          nxt[i-1][j]=nxt[i][j];
      nxt[i-1][P[i]-'a']=i;
  }
```

维护可持久化数组可以把时间复杂度降到 $O(n \lg |\alpha|)$,在字符集比较大的时候使用。

15.11.7.2 应用

下列序列数统计均指本质不同的序列。

子序列个数 记 $dp[i]$ 为从位置 i 开始的子序列个数,位置 i 的字符自成一个子序列,并且它与 $nxt[i][j]$ 位置的方案构成了本质不同的子序列,因此有

$$dp[i] = 1 + \sum_j dp[nxt[i][j]]$$

使用记忆化搜索或者逆序 dp。

公共子序列个数 例题「FJOI2016」所有公共子序列问题

预处理出两个字符串的序列自动机后,使用记忆化搜索在序列自动机上跑。

[10][后缀自动机]【学习笔记】https://www.cnblogs.com/candy99/p/6374177.html
[11]用 SAM 建广义后缀树 ≪ dwjshift's Blog http://dwjshift.logdown.com/posts/304570
[12]一个用 SAM 维护多个串的根号特技 https://www.cnblogs.com/mangoyang/p/10155185.html

回文子序列个数 对原串和反串构建序列自动机, 求这两个串的公共子序列数。

记记忆化搜索调用为 $DFS(x,y)$, x,y 分别为在这两个串上的匹配位置, 有 $x \leq n+1-y$, 等号成立意味着该回文序列为奇序列。但是奇序列不一定满足其等号成立, 如果记忆化搜索搜索到一个偶回文序列, 删掉该回文序列中心的一个字符, 就会出现新的奇回文序列, 因此在搜索时若满足 $x+y<n+1$, 要补上该奇序列的贡献。

上述内容参考了 pig_dog_baby 的博客[13]。

15.12 算术表达式解析

此类表达式由三种元素构成:操作数, 从左到右结合的二元运算符和括号。
解析算法如下:

- 实现简易的词法分析器区分出操作数、运算符和括号;
- 维护操作数栈和运算符栈(优先级单调递增), 扫到:
  - 操作数:将其压入操作数栈
  - 左括号:将其压入运算符栈
  - 运算符:取出不低于其优先级的运算符, 并取出对应的操作数进行运算, 将左操作数折叠完毕后将其压入运算符栈;
  - 右括号:取出从左括号到栈顶的运算符进行计算。
- 不断取出操作数和运算符更新栈, 直至运算符栈为空, 此时操作数栈只剩 1 个操作数(如果表达式合法), 该数即为结果。

实例:四则运算表达式解析(不支持一元运算符)

```
#include <cstdio>
#include <stack>
std::stack<int> num;
std::stack<char> op;
char buf[1024];
int pri[128];
bool fold() {
    if(num.size() < 2 || op.size() < 1)
        return false;
    int b = num.top();
    num.pop();
    int a = num.top();
    num.pop();
    char oper = op.top();
    op.pop();
    switch(oper) {
        case '+':
            num.push(a + b);
            break;
        case '-':
            num.push(a - b);
```

[13]序列自动机(一个数组而已...)及经典例题 https://blog.csdn.net/pig_dog_baby/article/details/81145857

```
            break;
        case '*':
            num.push(a * b);
            break;
        case '/':
            if(b) {
                num.push(a / b);
            } else
                return false;
            break;
    }
    return true;
}
template <typename T>
void clear(T& ct) {
    T empty;
    std::swap(ct, empty);
}
int main() {
    pri['+'] = pri['-'] = 1;
    pri['*'] = pri['/'] = 2;
    while(scanf("%s", buf) != EOF) {
        clear(num);
        clear(op);
        int i;
        for(i = 0; buf[i]; ++i) {
            char c = buf[i];
            if('0' <= c && c <= '9') {
                int number = 0;
                do {
                    number = number * 10 + c - '0';
                    c = buf[++i];
                } while('0' <= c && c <= '9');
                --i;
                num.push(number);
            } else if(c == '(')
                op.push('(');
            else if(c == ')') {
                bool flag = true;
                while(op.size() && op.top() != '(') {
                    if(!fold()) {
                        flag = false;
                        break;
                    }
                }
                if(flag && op.size())
                    op.pop();
                else
                    break;
            } else if(pri[c]) {
```

```
                bool flag = true;
                while(op.size() &&
                      pri[op.top()] >= pri[c]) {
                    if(!fold()) {
                        flag = false;
                        break;
                    }
                }
                if(flag)
                    op.push(c);
                else
                    break;
            } else
                break;
        }
        bool flag = true;
        while(op.size()) {
            if(!fold()) {
                flag = false;
                break;
            }
        }
        if(flag && !buf[i] && num.size() == 1)
            printf("%s=%d\n", buf, num.top());
        else
            puts("Error");
    }
    return 0;
}
```

15.13 Z Algorithm

给定一个字符串 P, 记 $Z(i)$ 为以 P 的第 i 个字符为首（从 0 开始）的后缀与 P 的 LCP 长度。

15.13.1 求解

核心思想类似于 Manacher 算法, 利用之前的计算结果尽可能减少暴力匹配操作。
算法步骤如下:

1. 首先有 $Z(0) = |P|$。
2. 维护当前已匹配的最右端, 即 $max\{i + Z(i) - 1\}$, 记为 R; 同时维护 R 对应的 i, 记为 L。
3. 考虑已经维护了当前 L, R 和 $Z[0 \cdots i-1]$, 求 $Z(i)$ 并更新 L, R。
   - 若 $i > R$, 直接从 i 开始暴力匹配, 然后令 $L = i, R = i + Z(i) - 1$;
   - 否则 $i \le R$, 那么有 $P[i - L \cdots R - L] = P[i \cdots R]$。然后有 $Z(i) \ge min(Z(i - L), R - i + 1)$。考虑 $Z(i - L)$ 与 $R - i + 1$ 的关系:

- 若 $Z(i-L)<R-i+1$, 则不必继续匹配, 令 $Z(i)=Z(i-L)$。
- 否则从 $Z(i)=R-i+1$ 开始暴力匹配, 更新 L,R。

模板:

```
int L=0,R=0;
for(int i=1;i<n;++i) {
    if(i>R) {
        L=R=i;
        while(R<n && P[R-L]==P[R])
            ++R;
        Z[i]=R-L;
        --R;
    }
    else {
        if(Z[i-L]<R-i+1)
            Z[i]=Z[i-L];
        else {
            L=i;
            while(R<n && P[R-L]==P[R])
                ++R;
            Z[i]=R-L;
            --R;
        }
    }
}
```

时间复杂度 $O(|P|)$。

15.13.2 应用

Z Algorithm 等价于 ExKMP。ExKMP 用来求解串 S 的每一个后缀 S_i 与另一个串 T 的 LCP 长度。

使用 Z Algorithm 可以解决: 构造新串 $T+$ 分隔符 $+S$, 运行 Z Algorithm, S 部分的 Z 值就是所求答案。实际上无需构造新串, 由于分隔符的存在, 可以分两次对 T 与 S 运行算法。

上述内容参考了 yashem66 的译文[14]。

15.14 卷积法解决字符串匹配问题

15.14.1 回文子序列

以某个位置为对称轴的回文子序列的个数可以由关于这个位置对称的字符对数计算。每一对都有选与不选两种选择, 除去全不选的情况, 记对称字符对数为 k, 方案为 2^k-1。

接下来考虑如何计算出对称字符对数。若字符串按照 Manacher 算法处理, 对于每个对称中心 i, 以它为对称中心的字符对满足 $S[i-x]=S[i+x]$, 注意到 $i-x+i+x=2i$ 为定值, 可以联系到卷积。枚举字符集的字符, 将有该字符的位置标为 1, 其余标为 0, 做一

[14]译文:Z-function/Z Algorithm 的构造与应用
https://blog.csdn.net/qq_33330876/article/details/72844491
原文:Z Algorithm
http://codeforces.com/blog/entry/3107

遍自卷积, 位置 $2i$ 的系数指示了以 i 为对称中心的当前字符对数。由于同一个位置上会被统计 1 次, 不同位置的对会被统计 2 次, 所以(系数 $+1$)$/2$ 才是实际对数。时间复杂度 $O(|\Sigma|n \lg n)$。

事实上卷积时不一定用 Manacher 算法预处理, 将偶回文序列的对称轴看做 $x.5$, 其两倍仍然是整数, 可直接统计 $[1, 2n]$ 全部系数。

15.14.1.1 例题

BZOJ3160: 万径人踪灭

本题要求的是回文子序列数, 去掉是连续一段的回文子串。回文子序列数可以使用 FFT 卷积或者序列自动机实现, 回文子串数可以用 Manacher 或者 PAM 实现。

参考代码(NTT+Manacher):

```
#include <algorithm>
#include <cstdio>
#include <cstring>
typedef long long Int64;
#define asInt64 static_cast<Int64>
Int64 powm(Int64 a, int k, Int64 mod) {
    Int64 res = 1;
    while(k) {
        if(k & 1)
            res = res * a % mod;
        k >>= 1, a = a * a % mod;
    }
    return res;
}
namespace Conv {
    const int size = 1 << 18, mod = 998244353;
    int tot, root[size], invR[size];
    int add(int a, int b) {
        a += b;
        return a < mod ? a : a - mod;
    }
    int sub(int a, int b) {
        a -= b;
        return a >= 0 ? a : a + mod;
    }
    void init(int n) {
        tot = n;
        Int64 base = powm(3, (mod - 1) / n, mod);
        Int64 invBase = powm(base, mod - 2, mod);
        root[0] = invR[0] = 1;
        for(int i = 1; i < n; ++i)
            root[i] = root[i - 1] * base % mod;
        for(int i = 1; i < n; ++i)
            invR[i] = invR[i - 1] * invBase % mod;
    }
    int cw[size];
    void NTT(int n, int* A, const int* w) {
```

```
        for(int i = 0, j = 0; i < n; ++i) {
            if(i < j)
                std::swap(A[i], A[j]);
            for(int l = n >> 1; (j ^= l) < l; l >>= 1)
                ;
        }
        for(int i = 2; i <= n; i <<= 1) {
            int m = i >> 1, fac = tot / i;
            for(int j = 0; j < m; ++j)
                cw[j] = w[j * fac];
            for(int j = 0; j < n; j += i)
                for(int k = 0; k < m; ++k) {
                    int &x = A[j + k],
                        &y = A[j + k + m];
                    int t = asInt64(y) * cw[k] % mod;
                    y = sub(x, t);
                    x = add(x, t);
                }
        }
    }
    int A[size];
    void selfConv(int n) {
        int p = 1;
        while(p <= n)
            p <<= 1;
        NTT(p, A, root);
        for(int i = 0; i < p; ++i)
            A[i] = asInt64(A[i]) * A[i] % mod;
        NTT(p, A, invR);
        Int64 div = powm(p, mod - 2, mod);
        for(int i = 1; i <= n; ++i)
            A[i] = A[i] * div % mod;
    }
}
const int maxn = 100005, mod = 1000000007;
char buf[maxn], bm[maxn * 2];
int R[maxn * 2];
int manacher(int n) {
    int cnt = 0;
    bm[cnt++] = '*';
    bm[cnt++] = '-';
    for(int i = 1; i <= n; ++i) {
        bm[cnt++] = buf[i];
        bm[cnt++] = '-';
    }
    int maxr = 0, mi = 0, res = 0;
    for(int i = 1; i < cnt; ++i) {
        R[i] = (maxr > i ?
                    std::min(maxr - i, R[2 * mi - i]) :
                    1);
```

```
        while(bm[i - R[i]] == bm[i + R[i]])
            ++R[i];
        if(maxr < i + R[i])
            maxr = i + R[i], mi = i;
        res += R[i] >> 1;
        if(res >= mod)
            res -= mod;
    }
    return res;
}
int C[maxn * 2], p2[maxn];
int main() {
    int c, n = 0;
    do
        c = getchar();
    while(c != 'a' && c != 'b');
    while(c == 'a' || c == 'b') {
        buf[++n] = c;
        c = getchar();
    }
    int end = n << 1, p = 1;
    while(p <= end)
        p <<= 1;
    Conv::init(p);
    for(int ch = 'a'; ch <= 'b'; ++ch) {
        memset(Conv::A, 0, sizeof(Conv::A));
        for(int i = 1; i <= n; ++i)
            Conv::A[i] = buf[i] == ch;
        Conv::selfConv(end);
        for(int i = 1; i <= end; ++i)
            C[i] += (Conv::A[i] + 1) >> 1;
    }
    int maxc = 0;
    for(int i = 1; i <= end; ++i)
        maxc = std::max(maxc, C[i]);
    p2[0] = 1;
    for(int i = 1; i <= maxc; ++i) {
        p2[i] = p2[i - 1] << 1;
        if(p2[i] >= mod)
            p2[i] -= mod;
    }
    int res = mod - end;
    for(int i = 1; i <= end; ++i) {
        res += p2[C[i]];
        if(res >= mod)
            res -= mod;
    }
    res = (res - manacher(n) + mod) % mod;
    printf("%d\n", res);
    return 0;
```

}

由于卷积出的值很小(在 n 的范围内), FFT、NTT 均可, 注意控制 FFT 的精度(做完除法操作后使用固定 eps, 如果想要省去除法操作, 需要将 eps 乘以 FFT 规模作为实际 eps)。

15.14.2 带通配符匹配

给定母串 S 与带通配符的模板串 P, 求 P 在 S 中的出现位置。

首先考虑不带通配符匹配的问题, 可以使用 KMP 解决(带通配符则无法保持 nxt 的性质), 但也有卷积的方法。考虑如何将其表示为卷积的形式。如果母串 S 在位置 i 处匹配了 P, 那么有 $S[i+k-1]=P[k], 1\le k\le |P|$。等式两边的下标之和不为定值, 但它们的差为定值。那么可以将 P 取反为 P_{rev}, 有 $S[i+k-1]=P_{rev}[|P|-k+1], 1\le k\le |P|$, 两边下标之和为 $i+|P|$, 可以进行卷积。同样考虑枚举字符集的字符, 将有该字符的位置置为 1, 其余置 0。将每次卷积的结果累加, 若位置 $i+|P|$ 上的系数为 $|P|$, 则说明母串 S 在位置 i 匹配上了 P。

有通配符的情况类似, 每个有通配符的位置强制置 1。

这种方法的时间复杂度仍为 $O(|\Sigma| n\lg n)$。

15.14.3 大字符集处理

对于 $|\Sigma|$ 较大的情况(比如 26 个字母), 26 次 DFT 的时间无法被接受。考虑如何把它们放在一个式子内计算。考虑不带通配符的情况, 将字母表示为数字, 对应位相等则数字差为 0。用区间内差的绝对值之和为 0 表示整段对应区间数字差为 0 比较麻烦, 索性使用平方和。那么有 $V[x]=\sum_{i=1}^{|P|}(S[x+i-1]-P[i])^2=0$, 将平方展开, P 取反得到

$$V[x]=\sum_{i=1}^{|P|}S[x+i-1]^2+P_{rev}[|P|-i+1]^2-2S[x+i-1]P_{rev}[|P|-i+1]$$

仅需做一次卷积。考虑带通配符的情况, 通配符无法表示为与 26 个数字都相等的数字, 但是可以令其为 0, 作为平方和的系数, 也可以使整个式子的值为 0。将式子拆开后可表示为两个卷积 + 一个常数的形式。如果母串也带通配符, 则再乘一个系数, 表示为三个卷积之和。

参考代码:

```
#include <algorithm>
#include <cmath>
#include <complex>
#include <cstdio>
#include <cstring>
typedef double FT;
#define asFT static_cast<FT>
typedef std::complex<FT> Complex;
const int size = 1 << 18;
Complex root[size], invR[size];
int tot;
void init(int n) {
    tot = n;
    FT base = 2.0 * acos(-1.0) / n;
```

```
    for(int i = 0; i < n; ++i) {
        root[i] =
            Complex(cos(i * base), sin(i * base));
        invR[i] = std::conj(root[i]);
    }
}
void FFT(int n, Complex* A, const Complex* w) {
    for(int i = 0, j = 0; i < n; ++i) {
        if(i < j)
            std::swap(A[i], A[j]);
        for(int l = n >> 1; (j ^= l) < l; l >>= 1)
            ;
    }
    for(int i = 2; i <= n; i <<= 1) {
        int m = i >> 1, fac = tot / i;
        for(int j = 0; j < n; j += i)
            for(int k = 0; k < m; ++k) {
                Complex &x = A[j + k],
                        &y = A[j + k + m];
                Complex t = y * w[k * fac];
                y = x - t;
                x += t;
            }
    }
}
const int maxn = 100005;
char S[maxn], T[maxn];
Complex A1[size], B1[size], A2[size], B2[size],
    C[size];
FT cube[128];
int pos[maxn];
int main() {
    scanf("%s%s", S + 1, T + 1);
    int lenS = strlen(S + 1), lenT = strlen(T + 1);
    std::reverse(T + 1, T + lenT + 1);
    for(int i = 'a'; i <= 'z'; ++i)
        cube[i] = i * i * i;
    FT cubeSum = 0.0;
    for(int i = 1; i <= lenT; ++i) {
        if(T[i] == '?')
            T[i] = 0;
        else
            cubeSum += cube[T[i]];
    }
    int p = 1;
    while(p <= (lenS + lenT))
        p <<= 1;
    cubeSum *= p;
    init(p);
```

```
    for(int i = 1; i <= lenS; ++i)
        A1[i] = asFT(S[i]) * S[i];
    for(int i = 1; i <= lenT; ++i)
        B1[i] = T[i];
    FFT(p, A1, root);
    FFT(p, B1, root);
    for(int i = 1; i <= lenS; ++i)
        A2[i] = S[i];
    for(int i = 1; i <= lenT; ++i)
        B2[i] = asFT(T[i]) * T[i];
    FFT(p, A2, root);
    FFT(p, B2, root);
    for(int i = 0; i < p; ++i)
        C[i] = A1[i] * B1[i] - 2.0 * A2[i] * B2[i];
    FFT(p, C, invR);
    int end = lenS - lenT + 1, pcnt = 0;
    FT eps = 0.5 * p;
    for(int i = 1; i <= end; ++i)
        if(fabs(C[i + lenT] + cubeSum) < eps)
            pos[++pcnt] = i - 1;
    printf("%d\n", pcnt);
    for(int i = 1; i <= pcnt; ++i)
        printf("%d\n", pos[i]);
    return 0;
}
```

使用 FFT 时，若最后不做除法，eps 要开大些，比如 0.5*p。可以在做点值乘法时直接求和，仅需一次 IDFT。

上述内容参考了小蒟蒻 yyb 的博客[15]。

15.14.4 广义模式匹配

例题：「THUPC2018」赛艇 / Citing

给定母方阵与模式方阵，求出所有匹配位置。

将母方阵按行拼接为一个串。模式方阵也如此拼接，但是行长要对齐到母方阵的行长便于匹配，不在模式方阵的部分填为通配符。如此将该问题转化为带通配符的字符串匹配问题。

注意有可能出现一些匹配位置导致模式方阵在母方阵上面展开后错位的情况，因为实际上这个匹配位置会导致模式方阵放上去后越界。为了避免这种情况，需要根据模式方阵的大小确定这个匹配位置是否合法。

有些题目还要求某些匹配位置的模式覆盖面积，同样可以将其转换为串，用模式起始位置与模式覆盖点卷积，最后可以得到每个点的覆盖次数。

15.15 最小表示法

最小表示法用来解决字符串的循环同构问题。一个字符串的最小表示就是它的字典序最小的循环串。如果两个字符串的循环表示相同，说明这两个字符串循环同构。

[15] [复习] 多项式和生成函数相关内容 https://www.cnblogs.com/cjyyb/p/10132855.html

考虑朴素算法:维护当前最小表示的起点 i 以及用于比较的起点 j, 初始 $i=0, j=1$。然后按照 $S[i]$ 与 $S[j]$ 的大小分类, 若 $S[i]=S[j]$, 则逐个比较直至 $S[i]\neq S[j]$; 若 $S[i]<S[j]$, 则说明起点 j 不可能成为答案, 令 j 后移; 若 $S[i]>S[j]$, 则说明 j 是更优的答案, 令 $i=j, j$ 后移。

上述算法的低效性在于 $S[i]=S[j]$ 的比较无法重复利用, 比如串 aaaaaa 可以将其卡到 $O(n^2)$。考虑记录当前起点 i 与 j 的前 k 位都相同, 当 $S[i+k]\neq S[j+k]$ 时, 需要移动某一个指针, 若只移动一位会导致下一次匹配时仍然重新匹配。那么需要增加指针跳跃的幅度。设 $S[i+k]>S[j+k]$, 那么 i 不是最优解, 由于 $S[i\dots i+k-1]=S[j\dots j+k-1]$, $(i,i+k]$ 范围内的起点也不是最优解, 因此 i 需要后移 $k+1$ 位。由于在当前阶段内已经扫描了 $k+1$ 次, 所以 i,j 的总偏移等于扫描次数, 由于总偏移不超过 $4n$, 算法的时间复杂度是 $O(n)$ 的。

算法退出的条件为 $i<n\wedge j<n\wedge k<n$, 若 i 或 $j\geq n$, 则说明扫描完毕, 另一个为合法解; 否则有 $k=n$, 两个均为最小表示。因此可以使用 $min(i,j)$ 作为最终解。

参考代码:

```
int scan(int n, const char* A) {
    int i = 0, j = 1, k = 0;
    while(i < n && j < n && k < n) {
        char ci = A[add(i, k, n)],
             cj = A[add(j, k, n)];
        if(ci == cj)
            ++k;
        else {
            (ci < cj ? j : i) += k + 1;
            if(i == j)
                ++j;
            k = 0;
        }
    }
    return std::min(i, j);
}
```

上述内容参考了 zy691357966 的博客[16]。

[16]字符串最小表示法 O(n) 算法 https://blog.csdn.net/zy691357966/article/details/39854359

Chapter 16

计算几何

16.1 基础设施 480
16.1.1 点,向量,直线,半平面的表示 480
16.1.2 点乘与叉乘 480
16.1.3 点到直线的距离 481
16.1.4 直线、线段的交点 481
16.1.5 判定点是否在多边形内 481
16.1.6 向量的旋转 482
16.1.7 坐标系的切换 482
16.1.8 点、向量、法向量的坐标变换 482
16.1.9 反射与折射 483
16.1.10 pick 定理 484
16.1.11 切比雪夫距离 484
16.1.12 精度处理 484
16.2 凸包 485
16.2.1 极角序凸包 485
16.2.2 水平序凸包 485
16.2.3 在线凸包 486
16.2.4 凸包矢量和(闵可夫斯基和) 486
16.2.5 凸包合并 486
16.2.6 稀疏包分布 486
16.2.7 二维最小乘积生成树 487
16.2.8 三维凸包 487
16.2.9 快速凸包 490
16.3 圆 495
16.3.1 圆的并 495
16.3.2 圆的交 502
16.3.3 最小圆覆盖 502
16.3.4 圆的反演 505
16.4 半平面交 505
16.4.1 基本算法 505
16.4.2 线性判空集 508
16.5 旋转卡壳 508
16.6 平面最近点对 509

16.1 基础设施

以下内容主要讨论二维空间中的计算几何。

16.1.1 点,向量,直线,半平面的表示

点与向量由 2 个坐标表示;半平面和直线由直线上一点 ori 与直线的方向向量 dir 表示;直线上的一点可表示为 $ori + dir * t$,通过控制参数 t 的取值还可以表示射线或线段;可以人为规定半平面的顺时针/逆时针 $180°$ 为半平面所在点集,通过叉积来判断点在半平面的哪一边。

```
typedef double FT;
struct Vec {
    FT x,y;
    //constructor
    //operator+-*
};
struct Line {
    Vec ori,dir;
    Vec operator()(FT t) const {
        return ori+dir*t;
    }
};
```

16.1.2 点乘与叉乘

16.1.2.1 点乘

向量点乘 $dot(a,b) = a.x * b.x + a.y * b.y = |a||b|cos < a,b >$,一般用来判断与法向量的夹角以及在某个向量上的投影长度。点乘满足加法分配律和交换律。

16.1.2.2 叉乘

向量叉乘 $cross(a,b) = a.x * b.y - b.x * a.y = |a||b|sin < a,b >$,这是两向量构成的平行四边形的有向面积。一般用来判断向量的相对方向以及计算多边形的面积。叉乘满足 $cross(a,b) = -cross(b,a)$ 和加法分配律(将其视作线性变换 $T : a \to cross(a,b)$ 或 $T : b \to cross(b,a)$ 可证)。

定理 16.1 (拉格朗日公式) *cross(a,cross(b,c))=b*dot(a,c)-c*dot(a,b)*

三维向量的叉乘计算了垂直于这两个向量的向量(两向量组成平面的法向量),即

$$cross(a,b) = \begin{pmatrix} a.y * b.z - b.y * a.z \\ a.z * b.x - b.z * a.x \\ a.x * b.y - b.x * a.y \end{pmatrix}$$

其方向满足右手定则(右手四指与大拇指垂直,食指指向向量 a,其余三指指向向量 b,大拇指方向即为叉乘方向),模长满足 $|cross(a,b)| = |a||b|sin < a,b >$。

16.1.2.3 体积计算

结合点乘和叉乘可得点 A 与点 B,C,D 所组成的三棱锥 $A-BCD$ 的有向体积为

$$V=\frac{1}{6}|dot(\overrightarrow{BA},cross(\overrightarrow{BC},\overrightarrow{BD}))|$$

其中 $\overrightarrow{N}=cross(\overrightarrow{BC},\overrightarrow{BD})$ 的方向为法向, 模长为底面面积的 2 倍。$|dot(\overrightarrow{BA},\overrightarrow{N})|$ 又给其模长增加了高的因子。套椎体体积公式可得上式。

16.1.2.4 极角计算

点乘可以得到 $x=|a||b|cos<a,b>$, 叉乘可以得到 $y=|a||b|sin<a,b>$, 将 (x,y) 看做在半径为 $|a||b|$ 的圆上的点, 极角为 $atan2(y,x)$。

16.1.3 点到直线的距离

设偏移向量 $delta=p-ori$:

- 计算 $dot(delta,dir)/|dir|$ 可得到投影长度 d', 根据勾股定理得到 $d^2=|delta|^2-d'^2$。
- 计算 $|cross(delta,dir)|$ 可得偏移向量与方向向量构成的平行四边形的面积, 根据面积公式得到 $d=|\frac{cross(delta,dir)}{|dir|}|$。

叉乘法的运算量少且精度较高, 能够指示半平面方向, 建议选用。

16.1.4 直线、线段的交点

```
0 Vec intersect(const Line& a,const Line& b) {
      Vec delta=a.ori-b.ori;
      FT t=cross(b.dir,delta)/cross(a.dir,b.dir);
      return a.ori+a.dir*t;
  }
```

证明留坑待补。

线段相交也是如此, 但首先要判断两线段是否相交。将该问题转换为两线段是否互相平分。设线段为 $a-b,c-d$, 首先判断 $a-b$ 平分 $c-d$, 即 c,d 分别位于 $a-b$ 两边, 有 $cross(c-a,b-a)*cross(d-a,b-a)\leq 0$, 同理对 $c-d$ 也做一遍。

16.1.5 判定点是否在多边形内

16.1.5.1 随机射线法

从点 P 开始随机引出一条射线, 计算其与多边形的边的交点个数, 若为奇数次则在多边形内。注意射线恰好经过点时要重新选择方向。

16.1.5.2 旋转角法

从点 P 与多边形的一个点开始, 不断旋转到下一个点, 直至转完一圈为止。此时若点 P 旋转了 $0°$, 则在多边形外; 若点 P 旋转了 $360°$, 则在多边形内。旋转角可以使用前文所述方法计算, 为了避免精度问题, 以 $\pi(180°)$ 为界进行比较。

16.1.5.3 半平面法

若该多边形为凸多边形, 以每条边构造半平面, 使用叉积判断是否在半平面内, 若点在所有半平面内则在多边形内。

16.1.6 向量的旋转

根据复数乘法的规律: 模长相乘, 幅角相加。构造逆时针旋转角度 θ 的单位旋转向量 $e^{\theta i} = \cos\theta + \sin\theta i$, 将原向量乘以该旋转向量得到结果: $(x\cos\theta - y\sin\theta, x\sin\theta + y\cos\theta)$。

有时对点坐标以某点为原点进行旋转变换, 然后按照 x 轴顺序贪心计算是一个不错的骗分方法。

16.1.7 坐标系的切换

有同一 d 维坐标空间中的点 P 与单位正交向量组 $b_1, b_2, \cdots, b_d$, 以该向量组为新坐标空间的基向量, 那么点 P 在新坐标空间的坐标值为它在这些向量上的投影长度。

事实上, 将点 P 在新坐标系上的坐标 P' 变换回旧坐标系意味着 P' 左乘矩阵 $T = b_1, b_2, \cdots, b_d$, 即 $TP' = P$。由于矩阵 T 为正交矩阵, 有 $T^{-1} = T^T$。因此 $P' = T^T P$, 即投影长度。

16.1.8 点、向量、法向量的坐标变换

可以使用一个矩阵 R^{d*d} 来表示 d 维空间中的旋转, 缩放, 坐标系切换, 引入齐次坐标(即给向量再加一维, 非透视投影时恒为 1)可支持平移(仅对于点的变换有意义)。

在三维空间下使用 $4 * 4$ 的矩阵来表示对坐标的变换。

16.1.8.1 平移

将坐标平移 (x, y, z):

$$\begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

16.1.8.2 旋转

以基于 z 轴旋转 θ 为例:

思路是对 x, y 轴坐标进行二维旋转, z 轴坐标不变。

$$\begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

16.1.8.3 缩放

对坐标分别缩放 x, y, z:

$$\begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

16.1.8.4 向量变换

注意平移不会影响向量的变换, 因此将矩阵截断为 3*3 矩阵 $T' = mat3(T)$。

16.1.8.5 法向量变换

若法向量 N 变换按照向量变换计算, 若遇到缩放则会发生变换后不垂直于变换后平面的情况。因为缩放矩阵的基向量不是单位向量。考虑一个与该法向量垂直的向量 V, 满足 $N^T \cdot V = 0$。那么对于变换后的两向量仍然要保持垂直, 即 $N^T T'^{-1} \cdot T'V = 0$, 对左边进行转置得到 $N' = (T'^{-1})^T N$。

16.1.9 反射与折射

以下的入射向量、出射向量与法向量均为单位向量。

16.1.9.1 反射

根据反射定律, 反射向量由水平方向的向量 I_x 减去垂直方向的向量 I_y。列出方程组:

$$\begin{aligned} I &= I_x + I_y \\ I_y &= N \cdot dot(I, N) \\ O &= I_x - I_y \end{aligned}$$

解得 $O = I - 2N \cdot dot(I, N)$。

16.1.9.2 折射

斯涅尔定律描述了折射率与角度的关系:

定理 16.2 $\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$

同样以法向量和切向量为基向量进行正交分解, 记 $\eta = \frac{\eta_1}{\eta_2}$, 有

$$\begin{aligned} I &= I_x + I_y \\ I_y &= N \cdot dot(I, N) \\ I_x &= \sin\theta_1 T \\ O &= O_x + O_y = \sin\theta_2 T - \cos\theta_2 N \end{aligned}$$

化简:

$$\begin{aligned} O &= \sin\theta_2 T - \cos\theta_2 N \\ &= \frac{\sin\theta_2}{\sin\theta_1} I_x - \cos\theta_2 N \\ &= \frac{\eta_1}{\eta_2}(I - dot(I, N) \cdot N) - \cos\theta_2 N \\ &= \eta \cdot I - (\eta dot(I, N) + \cos\theta_2) N \end{aligned}$$

代码如下, 注意全反射的情况(即 $\sin\theta_2$ 超出值域):

```
0 Vec refract(Vec I,Vec N,FT eta) {
    FT idn=dot(I,N);
    FT cosO2=1.0-eta*eta*(1.0-idn*idn);
    if(cosO2<0.0)return Vec();
    FT k=eta*idn+sqrt(cosO2);
    return I*eta-k*N;
}
```

上述内容参考了 Milo Yip 的文章[1]和 glm 库的代码[2]。

16.1.10 pick 定理

定理 16.3 (Pick's Theorem) 若格点多边形内的点数为 a，落在边上的点数为 b，则该多边形的面积为 $a-\frac{b}{2}+1$。

16.1.11 切比雪夫距离

切比雪夫距离是两点坐标之差绝对值的最大值。

分别考虑到原点曼哈顿距离和切比雪夫距离为 1 的点 P,Q，发现将 P 绕原点旋转 $45°$ 再缩放 $\sqrt{2}$ 倍后等于 Q。

因此 $P(x,y)\to Q(x+y,x-y),Q(x,y)\to P(\frac{x+y}{2},\frac{x-y}{2})$。~~忽然联想到 FWTxor~~。

对于到某点的曼哈顿距离 $\le k$ 的限制，可以变换坐标系将其转化为到该点的切比雪夫距离 $\le k$，由此将斜的正方形转化为与坐标轴对齐的正方形，更好统计。

16.1.12 精度处理

一般引入 $eps=1e-8$ 来避免精度问题。

常见问题：

- 判断两个值相等：$fabs(a-b)<eps$。
- 要输出 1.00 却输出 0.99 或者要输出 0.0 却输出-0.0：对正值 $+eps$，负值 $-eps$，$\pm eps$ 内强制为 0。
- 已知 $\sin\theta$ 求 $\cos\theta$：$sqrt(1.0-cosTheta*cosTheta)$ 可能会因为传入 $sqrt$ 的参数小于 0 而返回 nan，在调用数学函数前要 clamp 到定义域内或特判。
- 若乘积表示的范围越界，则可以使用对数加表示数乘。
- 技巧：若要维护 $std::set<FT>$，在比较器中引入 eps 自动完成去重工作。
- 多个量级相差巨大的浮点数相加减时要尽量使每次相加的两个数量级差不多。比如连加应该要从小到大累加。
- 二分时尽量固定迭代次数而不是用 eps 比较。
- 一些判定性问题可以使用模意义下的数代替浮点数解决。

为了辅助判断两个值的大小关系，引入一个符号函数 $sign$：

[1]用 C 语言画光(五)：折射 https://zhuanlan.zhihu.com/p/31127076

[2]glm/func_geometric.inl at master ·g-truc/glm ·GitHub https://github.com/g-truc/glm/blob/master/glm/detail/func_geometric.inl

```
int sign(FT x) {
    return (x>eps)-(x<-eps);
}
```

该内容参考了 Ac_smile 的博客[3]与 Oyking 的博客[4]。

16.2 凸包

16.2.1 极角序凸包

经典算法是 Graham 扫描法。算法步骤如下：

- 选择一个纵坐标最低的点（若有多个选横坐标最小）加入凸包，以此为原点按极角对其他点排序；
- 按照极角序加入每一个节点，保持凸包相邻 3 个节点的凸性质，注意三点在一条直线上时选择距离较远的点。

16.2.2 水平序凸包

极角序计算凸包容易由于 $atan2$ 的精度问题而造成错误，并且不易处理共线问题（始边要求从近到远，终边要求从远到近）。考虑对横坐标升序排序（若横坐标相等则对纵坐标升序比较，主要用于解决左右边缘出现竖线的问题，当然也可以使用旋转扰动法避免），分别计算其凸包的上凸壳和下凸壳，最后合并两部分。

代码如下（CCW）：

```
Vec P[size],C[size];
void convexHull(int n) {
    std::sort(P+1,P+n+1,[](const Vec& a,const Vec& b) {
        return a.x!=b.x?a.x<b.x:a.y<b.y;
    });
    int top=1;
    C[1]=P[1];
    for(int i=2;i<=n;++i) {
        while(top>=2 && cross(C[top]-C[top-1],
            P[i]-C[top-1])<eps)
            --top;
        C[++top]=P[i];
    }
    for(int i=n-1;i>=1;--i) {
        while(top>=2 && cross(C[top]-C[top-1],
            P[i]-C[top-1])<eps)
            --top;
        C[++top]=P[i];
    }
}
```

[3]计算几何中的精度问题
https://www.cnblogs.com/acsmile/archive/2011/05/09/2040918.html
[4]IO/ACM 中来自浮点数的陷阱（收集向）https://www.cnblogs.com/oyking/p/3959905.html

16.2.3 在线凸包

在线凸包即每次向点集中加入新点，维护当前凸包的某些信息。

经典思路是按照极角序（选择凸包内的定点作为基准点，因为凸包会越来越大，所以可以取前 3 个点的重心）将凸包上的点存储在 set 上。加入点时在 set 上查询极角序相邻的点组成的边，判断加入该点后新增的两条边是否是凸的，如果是凸的则继续更新左右两边，直至局部全为凸。**注意跨 x 负半轴的情况（极角为 $-pi$ 与 $+pi$ 附近的点相邻）**。

16.2.3.1 水平序在线凸包

同理维护上下凸壳，根据 dwjshift 的博客[5]所述，可以将其横纵坐标取负再求一次凸壳，因此只要考虑维护下凸壳。

16.2.4 凸包矢量和（闵可夫斯基和）

已知点集 A, B，求点集 $C = \{P_1 + P_2 | P_1 \in A \wedge P_2 \in B\}$ 的凸包。

显然该凸包与点集 A, B 凸包相加的凸包相同，容易想到预处理点集 A, B 的凸包后对将每对凸包上的点之和加入集合做凸包。在点随机分布的情况下，这种方法的时间复杂度为 $O(n \lg n)$，因为凸包的期望规模为 $\Theta(\lg n)$。但这种方法会被卡成 $O(n^2 \lg n)$，当所有点都在凸包上时（比如输入为一个圆）。

考虑优化对两个凸包做加法的过程，容易发现按照一个方向构造新凸包时，在原凸包选择相加点的顺序是单调的。因此可以使用双指针法维护当前选择的点，每次判断哪个凸包上的指针后移（类似归并排序），就能在 $O(n)$ 复杂度内完成合并。注意在两个候选点的对应的累加点与上一个累加点在一条直线上时，两个指针都向后移动 1 位，取这两个指针指向点之和。

事实上这正是旋转卡壳的应用之一。

16.2.5 凸包合并

合并后的凸包有一部分是两个凸包上点的连边，一部分是凸包上一段链。P_i, Q_j 连边当且仅当：

- P_i 与 Q_j 是并踵点对；
- $P_{i-1}, P_{i+1}, Q_{i-1}, Q_{i+1}$ 在 $P_i - Q_j$ 的同一侧。

使用旋转卡壳法可在线性时间内合并凸包。该方法参考了 ACMaker 的博客[6]。

如果凸包是使用水平序构造出来的，可以顺便保存凸包上点的顺序，合并时归并再做一遍水平序凸包。

16.2.6 稀疏包分布

- 若点在圆面上均匀分布，则凸包期望规模为 $\Theta(n^{1/3})$。
- 若点在凸多边形内部取得，凸包期望规模为 $\Theta(\lg n)$。
- 若点根据二维正态分布取得，凸包期望规模为 $\Theta(\sqrt{\lg n})$。

该内容来自算法导论 [4] 思考题 33-5。

[5]实现水平序动态凸包的小技巧 ≪ dwjshift's Blog http://dwjshift.logdown.com/posts/285072
[6]旋转卡壳——合并凸包 https://blog.csdn.net/ACMaker/article/details/3561150

16.2.7 二维最小乘积生成树

选择 $n-1$ 条边使得图连通, 最小化所选边第一权值和与第二权值和的乘积。

解法: 将第一权值和当做横坐标, 第二权值和当做纵坐标, 每一棵生成树对应一个点。首先分别求出第一权值和和第二权值和的 MST, 标记这两棵 MST 对应点 A, B。那么最优解肯定在以 A, B 为端点的下凸壳上, 考虑计算到 AB 距离最远的点 C, 用它更新答案。它肯定在凸壳上并且排除了大部分解。然后对 AC, CB 进行递归计算。

接下来讨论如何计算最远点 C:

首先由于 AB 长度固定, 可以转化为求 $S_{\triangle ABC}$ 的面积最大值。使用叉积可得

$$\begin{aligned} 2S_{\triangle ABC} &= cross(\overrightarrow{AC}, \overrightarrow{AB}) \\ &= (C.x - A.x) * (B.y - A.y) - (B.x - A.x) * (C.y - A.y) \\ &= C.x * (B.y - A.y) - C.y * (B.x - A.x) + Constant \end{aligned}$$

计算权值后可转化为计算最大生成树, 为了简便可将其系数取反同样求 MST。递归结束的条件为找不到满足要求的点(即最大面积不为正)。

最小乘积最大匹配也使用类似做法。对于高维情况, 将其改为求到超平面上最远距离。

事实上这是快速凸包算法的过程。

上述内容参考了空灰冰魂的博客[7]。

16.2.8 三维凸包

这里使用 $O(n^2)$ 增量法计算三维凸包。通过链表来维护面以便快速删除, 以及使用邻接矩阵维护点的有向连接对应的面的编号便于更新。

该算法的主要思想是检查并删除 "可视面", 然后将未封闭的边缘与新点连边重新构成封闭立体。

- 首先选择四个不共面的点, 组成四面体, 将面加入链表(注意顶点顺序朝外为 CCW。可以计算重心到该面的有向体积, 使其为负)。
- 将其余点逐个加入凸包。枚举每一个面, 若该面到新点的四面体的有向体积为正, 则删除原来的面, 并检查三条边的邻接面与其组成的四面体, 直到其有向体积为负, 才加入新面。

在实现时使用 $fid[a][b]$ 来维护边 $a \to b$ 对应的面, 注意边是有向的, 下面的代码人为规定为 CCW 序。

代码如下:

```
#include <algorithm>
#include <cmath>
#include <cstdio>
typedef double FT;
const FT eps = 1e-8;
struct Vec {
    FT x, y, z;
    Vec() {}
    Vec(FT x, FT y, FT z) : x(x), y(y), z(z) {}
    Vec operator+(const Vec& rhs) const {
        return Vec(x + rhs.x, y + rhs.y, z + rhs.z);
```

[7]【BZOJ2395】【Balkan 2011】Timeismoney 最小乘积生成树 https://blog.csdn.net/vmurder/article/details/46828379

```
    }
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y, z - rhs.z);
    }
    Vec operator*(FT rhs) const {
        return Vec(x * rhs, y * rhs, z * rhs);
    }
};
Vec cross(const Vec& a, const Vec& b) {
    return Vec(a.y * b.z - b.y * a.z,
               a.z * b.x - b.z * a.x,
               a.x * b.y - b.x * a.y);
}
FT dot(const Vec& a, const Vec& b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
FT length(const Vec& a) {
    return sqrt(a.x * a.x + a.y * a.y + a.z * a.z);
}
FT area2(const Vec& a, const Vec& b, const Vec& c) {
    return length(cross(b - a, c - a));
}
FT vol6(const Vec& a, const Vec& b, const Vec& c,
        const Vec& d) {
    Vec ab = b - a, ac = c - a, ad = d - a;
    return dot(cross(ab, ac), ad);
}
const int size = 2005;
Vec P[size];
struct Face {
    int a, b, c;
    Face() {}
    Face(int a, int b, int c) : a(a), b(b), c(c) {}
    FT area2() const {
        return ::area2(P[a], P[b], P[c]);
    }
    FT vol6(int id) const {
        return ::vol6(P[a], P[b], P[c], P[id]);
    }
};
Face face[6 * size];
int fcnt = 0, pre[6 * size] = {}, nxt[6 * size] = {},
    fid[size][size] = {}, tail = 0;
void add(const Face& f) {
    int id = ++fcnt;
    fid[f.a][f.b] = fid[f.b][f.c] = fid[f.c][f.a] = id;
    nxt[tail] = id;
    pre[id] = tail;
    tail = id;
    face[id] = f;
```

```
    }
    void erase(int id) {
        Face f = face[id];
        fid[f.a][f.b] = fid[f.b][f.c] = fid[f.c][f.a] = 0;
        if(id == tail)
            tail = pre[id];
        int p = pre[id], n = nxt[id];
        nxt[p] = n, pre[n] = p;
    }
70  void erase(int i, int fi);
    void testEdge(int i, int a, int b) {
        int id = fid[a][b];
        if(id) {
            if(face[id].vol6(i) > eps)
                erase(i, id);
            else
                add(Face(b, a, i));
        }
    }
80  void erase(int i, int fi) {
        erase(fi);
        testEdge(i, face[fi].b, face[fi].a);
        testEdge(i, face[fi].c, face[fi].b);
        testEdge(i, face[fi].a, face[fi].c);
    }
    FT foo() {
        int n;
        scanf("%d", &n);
        if(n < 4)
90          return 0.0;
        for(int i = 0; i < n; ++i)
            scanf("%lf%lf%lf", &P[i].x, &P[i].y, &P[i].z);

        int flag = 0;
        for(int i = 2; i < n; ++i)
            if(fabs(area2(P[0], P[1], P[i])) > eps) {
                std::swap(P[2], P[i]);
                ++flag;
                break;
100         }
        if(flag != 1)
            return 0.0;
        for(int i = 3; i < n; ++i)
            if(fabs(vol6(P[0], P[1], P[2], P[i])) > eps) {
                std::swap(P[3], P[i]);
                ++flag;
                break;
            }
        if(flag != 2)
110         return 0.0;
```

```
    P[n] = (P[0] + P[1] + P[2] + P[3]) * 0.25;
    for(int i = 0; i < 4; ++i) {
        Face f(i, (i + 1) & 3, (i + 2) & 3);
        if(f.vol6(n) > 0.0)
            std::swap(f.a, f.c);
        add(f);
    }

    for(int i = 4; i < n; ++i) {
        for(int j = nxt[0]; j; j = nxt[j])
            if(face[j].vol6(i) > eps) {
                erase(i, j);
                break;
            }
    }
    FT res = 0.0;
    for(int i = nxt[0]; i; i = nxt[i])
        res += face[i].area2();
    return 0.5 * res;
}
int main() {
    printf("%.3lf\n", foo());
    return 0;
}
```

该方法参考了 _sunshine 的博客[8]。

16.2.9 快速凸包

高维凸包不易理解且使用增量法的 $O(n^2)$ 复杂度较大, 所以在此介绍快速凸包算法。在平均情况下复杂度为 $O(n \lg n)$, 最坏情况 $O(n^2)$, 与快速排序类似。

主要思想是每次选择到超平面的最远点, 该点肯定在凸包上, 并且该点与超平面将整个集合分割为 $d+1$ 个子集, 其中该点与超平面组成的体内的点直接被排除, 以达到快速求解的目的。

构造初始超平面时先选择一个点, 再选择离这个点最远的点构成线, 再选择离这条线最远的点构成三角平面, 以此类推。构造线也可以使用两个坐标为极值的点。还有一种随机化方法: 不断随机构造一个超平面, 计算到这个超平面的有向距离的最大值与最小值, 取出对应的点, 直到选出 $d+1$ 个不重复的点(二维情况下只需选取 2 个初始点, 三维情况则必须选取 4 个初始点)。

二维凸包代码:

```
// P2742
#include <algorithm>
#include <cmath>
#include <cstdio>
const int size = 10005;
typedef double FT;
```

[8]hdu 4266 三维凸包 (增量法) https://www.cnblogs.com/-sunshine/archive/2012/08/25/2656794.html

```
const FT inf = 1e20, eps = 1e-8;
struct Vec {
    FT x, y;
    Vec() {}
    Vec(FT x, FT y) : x(x), y(y) {}
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y);
    }
} P[size], latest;
FT dis(Vec a, Vec b) {
    FT dx = a.x - b.x, dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
FT cross(Vec a, Vec b) {
    return a.x * b.y - a.y * b.x;
}
struct Line {
    Vec ori, dst, dir;
    Line(Vec beg, Vec end)
        : ori(beg), dst(end), dir(end - beg) {}
    FT test(Vec p) const {
        return cross(p - ori, dir);
    }
};
FT ans = 0.0;
void quickHullImpl(int beg, int end, const Line& ab) {
    if(beg == end)
        return;
    FT marea = 0.0;
    Vec cv;
    for(int i = beg; i < end; ++i) {
        FT cc = ab.test(P[i]);
        if(cc > marea)
            marea = cc, cv = P[i];
    }
    int le = beg, rb = end;
    Line ac(ab.ori, cv), cb(cv, ab.dst);
    for(int i = beg; i < rb;) {
        if(cb.test(P[i]) > eps) {
            std::swap(P[--rb], P[i]);
        } else {
            if(ac.test(P[i]) > eps)
                P[le++] = P[i];
            ++i;
        }
    }
    quickHullImpl(beg, le, ac);
    ans += dis(latest, cv);
    latest = cv;
    quickHullImpl(rb, end, cb);
```

```
}
void quickHull(int n) {
    Vec lv = P[0], rv = P[0];
    for(int i = 1; i < n; ++i) {
        if(P[i].x < lv.x)
            lv = P[i];
        else if(P[i].x > rv.x)
            rv = P[i];
    }
    int le = 0, rb = n;
    Line ab(lv, rv), ba(rv, lv);
    for(int i = 0; i < rb;) {
        FT cv = ab.test(P[i]);
        if(cv < -eps)
            std::swap(P[--rb], P[i]);
        else {
            if(cv > eps)
                P[le++] = P[i];
            ++i;
        }
    }
    latest = lv;
    quickHullImpl(0, le, ab);
    ans += dis(latest, rv);
    latest = rv;
    quickHullImpl(rb, n, ba);
    ans += dis(latest, lv);
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n; ++i)
        scanf("%lf%lf", &P[i].x, &P[i].y);
    quickHull(n);
    printf("%.2lf\n", ans);
    return 0;
}
```

三维凸包代码(不正确):

```
// P4724
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <iostream>
const int size = 2005;
typedef long double FT;
const FT eps = 1e-10;
FT getOffset() {
    static int seed = 54343;
    seed = seed * 48271LL % 2147483647;
```

```
    return seed / 2147483647.0 * 1e-12L;
}
struct Vec {
    FT x, y, z;
    Vec() {}
    Vec(FT x, FT y, FT z) : x(x), y(y), z(z) {}
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y, z - rhs.z);
    }
    Vec operator+(const Vec& rhs) const {
        return Vec(x + rhs.x, y + rhs.y, z + rhs.z);
    }
    Vec operator*(FT k) const {
        return Vec(x * k, y * k, z * k);
    }
} P[size];
Vec cross(const Vec& a, const Vec& b) {
    return Vec(a.y * b.z - b.y * a.z,
               a.z * b.x - b.z * a.x,
               a.x * b.y - b.x * a.y);
}
FT dot(const Vec& a, const Vec& b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
FT length2(const Vec& a) {
    return a.x * a.x + a.y * a.y + a.z * a.z;
}
struct Plane {
    Vec a, b, c, N;
    Plane() {}
    Plane(const Vec& a, const Vec& b, const Vec& c)
        : a(a), b(b), c(c), N(cross(b - a, c - a)) {}
    FT area2() const {
        return sqrt(length2(N));
    }
    FT vol6(const Vec& d) const {
        Vec ad = d - a;
        return dot(N, ad);
    }
};
FT ans = 0.0;
void quickHullImpl(int beg, int end, const Plane& abc,
                   const Vec& d);
int partition(int beg, int end, const Plane& abc) {
    int mid = beg;
    FT maxv = 0.0;
    Vec d;
    for(int i = beg; i < end; ++i) {
        FT cc = abc.vol6(P[i]);
        if(cc > eps) {
```

```
            if(cc > maxv)
                d = P[i], maxv = cc;
            std::swap(P[i], P[mid++]);
        }
    }
    if(beg == mid)
        ans += abc.area2();
    else
        quickHullImpl(beg, mid, abc, d);
    return mid;
}
void quickHullImpl(int beg, int end, const Plane& abc,
                   const Vec& d) {
    beg = partition(beg, end, Plane(abc.a, abc.b, d));
    beg = partition(beg, end, Plane(abc.a, d, abc.c));
    partition(beg, end, Plane(abc.b, abc.c, d));
}
void quickHull(int n) {
    Vec lv = P[0], rv = P[0];
    for(int i = 1; i < n; ++i) {
        if(P[i].x < lv.x)
            lv = P[i];
        else if(P[i].x > rv.x)
            rv = P[i];
    }
    FT maxv = eps;
    Vec cv;
    for(int i = 0; i < n; ++i) {
        Plane cp(lv, rv, P[i]);
        FT cc = length2(cp.N);
        if(cc > maxv)
            maxv = cc, cv = P[i];
    }
    Plane bp(lv, rv, cv);
    maxv = eps;
    for(int i = 0; i < n; ++i) {
        FT cc = fabs(bp.vol6(P[i]));
        if(cc > maxv)
            maxv = cc, cv = P[i];
    }
    Vec ps[4] = { lv, rv, bp.c, cv };
    Vec mid = (lv + rv + bp.c + cv) * 0.25;
    int beg = 0;
    for(int i = 0; i < 4; ++i) {
        Vec np[3];
        int cnt = 0;
        for(int j = 0; j < 4; ++j)
            if(i != j)
                np[cnt++] = ps[j];
        Plane tp(np[0], np[1], np[2]);
```

```
            if(tp.vol6(mid) > eps)
                tp = Plane(np[0], np[2], np[1]);
            beg = partition(beg, n, tp);
        }
    }
    int main() {
        freopen("testdata.in", "r", stdin);
        int n;
        std::cin >> n;
        for(int i = 0; i < n; ++i) {
            std::cin >> P[i].x >> P[i].y >> P[i].z;
            P[i].x += getOffset();
            P[i].y += getOffset();
            P[i].z += getOffset();
        }
        quickHull(n);
        std::cout.precision(3);
        std::cout << std::fixed << ans * 0.5;
        return 0;
    }
```

计算凸包/表面积时四点共面的情况不太好处理。为了处理多于 3 点共面的情况, 可以给每个点的坐标值加一些 “扰动”。

三维凸包实现参见 Valve Software 的 Dirk Gregorius 在 GDC2014 上的文章 Implementing QuickHull[9]和 lloyd 的代码[10]。

16.2.9.1 快速凸包的精度控制

根据 Dirk Gregorius 的文章, ε 应为 $d * \varepsilon_{machine} * max\{max_i - min_i\}$。

上述内容参考了 Wikipedia-EN[11]。

16.3 圆

处理圆的交并问题一般考虑圆之间的覆盖区间。

16.3.1 圆的并

求圆的并的面积。

首先去除被其他圆覆盖的圆。然后对于每个圆与其它圆求交点, 得到每个圆被覆盖的弧度区间(逆时针为正方向)。注意跨越弧度 π 的区间要分为两个区间。对每个圆的覆盖区间排序得到不覆盖的区间。

画图可以发现圆并的面积等于圆弧面积 + 多边形的有向面积。由于多边形需要计算有向面积(直接以原点作为基准点), 不能直接计算扇形面积和。

对于半径为 R 的圆, 弧度为 x 的圆弧的面积为 $\frac{1}{2}(x - \sin x)R^2$。当 $x > \pi$ 时该等式也满足。

[9] PowerPoint Presentation - DirkGregorius_ImplementingQuickHull.pdf
http://media.steampowered.com/apps/valve/2014/DirkGregorius_ImplementingQuickHull.pdf

[10] QuickHull3D: A Robust 3D Convex Hull Algorithm in Java
https://www.cs.ubc.ca/~lloyd/java/quickhull3d.html

[11] Quickhull - Wikipedia https://en.wikipedia.org/wiki/Quickhull

两圆求交时用圆心、交点弦中点、某一交点构造两个直角三角形，然后根据勾股定理列出方程组，解出一个圆心到交点弦中点的距离，进而解出该三角形各边长，使用向量偏移计算交点弦中点和交点。

时间复杂度 $O(n^2 \lg n)$。

代码如下：

```
// SP8073
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <vector>
typedef double FT;
const FT eps = 1e-8, pi = acos(-1.0);
struct Vec {
    FT x, y;
    Vec() {}
    Vec(FT x, FT y) : x(x), y(y) {}
    Vec operator+(const Vec& rhs) const {
        return Vec(x + rhs.x, y + rhs.y);
    }
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y);
    }
    Vec operator*(FT k) const {
        return Vec(x * k, y * k);
    }
    Vec operator/(FT k) const {
        return Vec(x / k, y / k);
    }
    FT length2() const {
        return x * x + y * y;
    }
    FT angle() const {
        return atan2(y, x);
    }
};
FT dis(const Vec& a, const Vec& b) {
    return sqrt((a - b).length2());
}
FT cross(const Vec& a, const Vec& b) {
    return a.x * b.y - b.x * a.y;
}
struct Range {
    FT beg, end;
    Range(FT beg, FT end) : beg(beg), end(end) {}
    bool operator<(const Range& rhs) const {
        return fabs(beg - rhs.beg) > eps ?
            beg < rhs.beg :
            end < rhs.end;
    }
```

```
};
const int size = 1005;
struct Circle {
    Vec p;
    FT r;
    std::vector<Range> range;
    bool operator<(const Circle& rhs) const {
        return r < rhs.r;
    }
    FT area2(FT x) const {
        return r * r * (x - sin(x));
    }
    bool test(const Circle& rhs) const {
        FT dr = rhs.r - r;
        return (rhs.p - p).length2() <= dr * dr + eps;
    }
    Vec operator()(FT angle) const {
        return p + Vec(cos(angle) * r, sin(angle) * r);
    }
} C[size];
int filtrate(int n) {
    std::sort(C, C + n);
    int cnt = 0;
    for(int i = 0; i < n; ++i) {
        bool flag = true;
        for(int j = i + 1; j < n; ++j)
            if(C[i].test(C[j])) {
                flag = false;
                break;
            }
        if(flag)
            C[cnt++] = C[i];
    }
    return cnt;
}
void addRange(Circle& c, const Vec& beg,
              const Vec& end) {
    FT ba = (beg - c.p).angle(),
       ea = (end - c.p).angle();
    if(fabs(ea - ba) > eps) {
        if(ea > ba)
            c.range.push_back(Range(ba, ea));
        else {
            c.range.push_back(Range(ba, pi));
            c.range.push_back(Range(-pi, ea));
        }
    }
}
void intersect(Circle& a, Circle& b) {
    FT dab = dis(a.p, b.p);
```

```
        if(dab > a.r + b.r - eps)
            return;
        FT dac = (dab * dab + a.r * a.r - b.r * b.r) /
            (2.0 * dab);
        FT dec = sqrt(a.r * a.r - dac * dac);
        Vec base = (b.p - a.p) / dab;
        Vec c = a.p + base * dac;
        Vec off(-base.y * dec, base.x * dec);
        Vec p1 = c + off, p2 = c - off;
        addRange(a, p2, p1);
        addRange(b, p1, p2);
    }
    int main() {
        int n;
        scanf("%d", &n);
        for(int i = 0; i < n; ++i)
            scanf("%lf%lf%lf", &C[i].p.x, &C[i].p.y,
                  &C[i].r);
        n = filtrate(n);
        for(int i = 0; i < n; ++i)
            for(int j = i + 1; j < n; ++j)
                intersect(C[i], C[j]);
        FT ans = 0.0;
        for(int i = 0; i < n; ++i) {
            if(C[i].range.size()) {
                std::vector<Range>& r = C[i].range;
                std::sort(r.begin(), r.end());
                int cnt = 1;
                for(int j = 1; j < r.size(); ++j)
                    if(r[cnt - 1].end > r[j].beg - eps)
                        r[cnt - 1].end = std::max(
                            r[cnt - 1].end, r[j].end);
                    else
                        r[cnt++] = r[j];
                if(r.size() == cnt)
                    r.push_back(r[0]);
                else
                    r[cnt] = r[0];
                for(int j = 0; j < cnt; ++j) {
                    Vec beg = C[i](r[j].end),
                        end = C[i](r[j + 1].beg);
                    ans += cross(beg, end);
                    FT da = r[j + 1].beg - r[j].end;
                    if(da < -eps)
                        da += 2.0 * pi;
                    ans += C[i].area2(da);
                }
            } else
                ans += 2.0 * pi * C[i].r * C[i].r;
        }
```

```
    printf("%.3lf\n", 0.5 * ans);
    return 0;
}
```

16.3.1.1 扩展

求恰好被 i 个圆覆盖的区域面积。

同样对于每个圆考虑其覆盖区间, 发现常规算法求出的被覆盖 i 次的区域对被覆盖 $< i$ 次的区域有贡献, 因此计算完毕后要差分输出以扣除多余面积。对覆盖分界点排序, 每个区间差分标记覆盖, 使用前缀和计算统计区间覆盖次数, 计算贡献。**注意覆盖次数要加上整个圆被覆盖的次数。**

```
// SP8119
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
#include <vector>
typedef double FT;
const FT eps = 1e-8, pi = acos(-1.0);
struct Vec {
    FT x, y;
    Vec() {}
    Vec(FT x, FT y) : x(x), y(y) {}
    Vec operator+(const Vec& rhs) const {
        return Vec(x + rhs.x, y + rhs.y);
    }
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y);
    }
    Vec operator*(FT k) const {
        return Vec(x * k, y * k);
    }
    Vec operator/(FT k) const {
        return Vec(x / k, y / k);
    }
    FT length2() const {
        return x * x + y * y;
    }
    FT angle() const {
        return atan2(y, x);
    }
};
FT dis(const Vec& a, const Vec& b) {
    return sqrt((a - b).length2());
}
FT cross(const Vec& a, const Vec& b) {
    return a.x * b.y - b.x * a.y;
}
struct Range {
```

```
    FT beg, end;
    Range(FT beg, FT end) : beg(beg), end(end) {}
};
const int size = 1005;
struct Circle {
    Vec p;
    FT r;
    int c;
    std::vector<Range> range;
    bool operator<(const Circle& rhs) const {
        return r < rhs.r;
    }
    FT area2(FT x) const {
        return r * r * (x - sin(x));
    }
    Vec operator()(FT angle) const {
        return p + Vec(cos(angle) * r, sin(angle) * r);
    }
} C[size];
void addRange(Circle& c, const Vec& beg,
              const Vec& end) {
    FT ba = (beg - c.p).angle(),
       ea = (end - c.p).angle();
    if(fabs(ea - ba) > eps) {
        if(ea > ba)
            c.range.push_back(Range(ba, ea));
        else {
            c.range.push_back(Range(ba, pi));
            c.range.push_back(Range(-pi, ea));
        }
    }
}
void intersect(Circle& a, Circle& b) {
    FT dab = dis(a.p, b.p);
    if(dab > a.r + b.r - eps)
        return;
    if(dab < fabs(a.r - b.r) + eps) {
        if(a.r - b.r > -eps)
            ++b.c;
        if(b.r - a.r > -eps)
            ++a.c;
        return;
    }
    FT dac = (dab * dab + a.r * a.r - b.r * b.r) /
        (2.0 * dab);
    FT dec = sqrt(a.r * a.r - dac * dac);
    Vec base = (b.p - a.p) / dab;
    Vec c = a.p + base * dac;
    Vec off(-base.y * dec, base.x * dec);
    Vec p1 = c + off, p2 = c - off;
```

```
        addRange(a, p2, p1);
        addRange(b, p1, p2);
90  }
    FT ans[size], mr[size * 2];
    int sum[size * 2];
    bool cmpEq(FT a, FT b) {
        return fabs(a - b) < eps;
    }
    bool cmpLe(FT a, FT b) {
        return b - a > eps;
    }
    int find(int siz, FT x) {
100     return std::lower_bound(mr, mr + siz, x, cmpLe) -
            mr;
    }
    int main() {
        int n;
        scanf("%d", &n);
        for(int i = 0; i < n; ++i)
            scanf("%lf%lf%lf", &C[i].p.x, &C[i].p.y,
                  &C[i].r);
        for(int i = 0; i < n; ++i)
110         for(int j = i + 1; j < n; ++j)
                intersect(C[i], C[j]);
        for(int i = 0; i < n; ++i) {
            if(C[i].range.size()) {
                std::vector<Range>& r = C[i].range;
                int cnt = 2;
                mr[0] = -pi, mr[1] = pi;
                for(int j = 0; j < r.size(); ++j) {
                    mr[cnt++] = r[j].beg;
                    mr[cnt++] = r[j].end;
120             }
                memset(sum, 0, sizeof(int) * cnt);
                std::sort(mr, mr + cnt);
                cnt =
                    std::unique(mr, mr + cnt, cmpEq) - mr;
                for(int j = 0; j < r.size(); ++j) {
                    ++sum[find(cnt, r[j].beg)];
                    --sum[find(cnt, r[j].end)];
                }
                int c = C[i].c;
130             for(int j = 1; j < cnt; ++j) {
                    FT b = mr[j - 1], e = mr[j];
                    c += sum[j - 1];
                    Vec beg = C[i](b), end = C[i](e);
                    ans[c] += cross(beg, end);
                    ans[c] += C[i].area2(e - b);
                }
            } else
```

```
            ans[C[i].c] += 2.0 * pi * C[i].r * C[i].r;
    }
    for(int i = 0; i < n; ++i)
        printf("[%d] = %.3lf\n", i + 1,
               0.5 * (ans[i] - ans[i + 1]));
    return 0;
}
```

上述内容参考了 Oyking 的博客[12]。

16.3.2 圆的交

与求圆并同理, 对于每个圆求出覆盖区间的交, 答案贡献为圆弧面积 $+\frac{1}{2}$ 区间交两端点的叉积。

16.3.3 最小圆覆盖

一般使用随机增量法:

1. 将输入点随机重排列;
2. 构造一个初始空圆(退化为点且没有点在这个点上);
3. 不断加入点更新当前最小覆盖圆, 设当前点为 P_i:
   (a) 若该点已经在该圆内, 跳出;
   (b) 否则该点必在新圆上。将当前圆重置为一个空圆, 固定该圆必有点 P_i。按顺序加入点 $P_j, j < i$, 若 P_j 不在当前圆内, 则 P_j 也在新圆上。于是将当前圆重置为以 P_i, P_j 为直径的圆, 找出不在当前圆上的点 $P_k, k < j$, 则 P_i, P_j, P_k 三点可确定一个圆, 保证该圆是 $P_i, P_j, P_x, x \le k$ 的最小覆盖圆, 更新当前圆后继续迭代。

看似时间复杂度为 $O(n^3)$, 事实上该算法的时间复杂度为 $O(n)$。

实现细节 在求三角形外接圆时注意对三点共线的点取最远点对作为直径, 三点不共线则使用中垂线求交求外接圆。注意中垂线求交时要选用夹角较大的一对求交, 并且对半径取 max。

Update:其实不必考虑三点共线的情况, 这种情况已经被之前的两点共线考虑过了, 而求出来的外接圆不会更优。

代码如下:

```
// P1742
#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstdlib>
const int size = 100005;
typedef double FT;
const FT eps = 1e-12;
FT read() {
```

[12]SPOJ 8073 The area of the union of circles (计算几何の圆并)(CIRU) https://www.cnblogs.com/oyking/p/3424999.html

```
    char buf[32];
    int cnt = 0, c;
    do
        c = getchar();
    while(!isgraph(c));
    while(isgraph(c)) {
        buf[cnt++] = c;
        c = getchar();
    }
    buf[cnt] = '\0';
    return strtod(buf, 0);
}
struct Vec {
    FT x, y;
    Vec() {}
    Vec(FT x, FT y) : x(x), y(y) {}
    Vec operator+(const Vec& rhs) const {
        return Vec(x + rhs.x, y + rhs.y);
    }
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y);
    }
    Vec operator*(FT k) const {
        return Vec(x * k, y * k);
    }
    FT length2() const {
        return x * x + y * y;
    }
} P[size];
FT cross(const Vec& a, const Vec& b) {
    return a.x * b.y - b.x * a.y;
}
struct Circle {
    Vec c;
    FT r2;
    Circle(const Vec& c, FT r2) : c(c), r2(r2 + eps) {}
    bool out(const Vec& p) const {
        return (p - c).length2() > r2;
    }
};
Circle makeCircle(const Vec& a, const Vec& b) {
    Vec mid = (a + b) * 0.5;
    return Circle(mid, (a - mid).length2());
}
struct Line {
    Vec ori, dir;
    Line(const Vec& a, const Vec& b) {
        ori = (a + b) * 0.5;
        Vec delta = a - b;
        dir = Vec(-delta.y, delta.x);
```

```
    }
};
Vec intersect(const Line& a, const Line& b) {
    Vec delta = a.ori - b.ori;
    FT t = cross(b.dir, delta) / cross(a.dir, b.dir);
    return a.ori + a.dir * t;
}
Circle makeCircle(const Vec& a, const Vec& b,
                  const Vec& c) {
    Line ab(a, b), ac(a, c);
    if(fabs(cross(ab.dir, ac.dir)) < eps) {
        FT dab = (a - b).length2(),
           dac = (a - c).length2(),
           dbc = (b - c).length2();
        FT maxd = fmax(dab, fmax(dac, dbc));
        if(maxd == dab)
            return makeCircle(a, b);
        if(maxd == dac)
            return makeCircle(a, c);
        return makeCircle(b, c);
    }
    Vec o = intersect(ab, ac);
    return Circle(o, (a - o).length2());
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n; ++i) {
        P[i].x = read();
        P[i].y = read();
    }
    std::random_shuffle(P, P + n);
    Circle c(Vec(1e10, 1e10), 0.0);
    for(int i = 0; i < n; ++i) {
        if(c.out(P[i])) {
            c = Circle(P[i], 0.0);
            for(int j = 0; j < i; ++j) {
                if(c.out(P[j])) {
                    c = makeCircle(P[i], P[j]);
                    for(int k = 0; k < j; ++k) {
                        if(c.out(P[k]))
                            c = makeCircle(P[i], P[j],
                                           P[k]);
                    }
                }
            }
        }
    }
    printf("%.10lf\n%.10lf %.10lf\n", sqrt(c.r2),
           c.c.x, c.c.y);
```

```
110     return 0;
}
```

上述内容参考了 Wikipedia-EN[13]。

16.3.3.1 高维最小超球覆盖

例题:LOJ#6360. 复燃「恋之埋火」

可以沿用二维的情况进行随机增量法(DFS), 关键在于如何根据给定的 k 个点生成 m 维超球, 且圆心在这 k 个点组成的 $k-1$ 维平面上。

由于圆心在这 k 个点组成的 $k-1$ 维平面上, 可以使用 $k-1$ 个向量组成线性基来表示圆心。那么可以以 P_0 作为原点, $D_i = P_i - P_0$ 作为基向量。此时圆心被表示为这些基向量的线性加权和, 可以列出线性方程组。设圆心坐标为 $O = P_0 + Ax$, A 是由基向量组成的矩阵, 由于圆心到这些点的距离相等, 有 $(Ax)^2 = (Ax - D_i)^2$, 展开化简得 $2D_i^T Ax = D_i^2$, 高斯消元解出 x, 再计算出 O。

同理此题也不必考虑高斯消元无解的情况。

16.3.4 圆的反演

定义 已知 $\odot O$ 的半径为 r, 若点 P_1, P_2 在以点 O 为端点的射线上, 且 $OP_1 \cdot OP_2 = r^2$, 则称 P_1, P_2 关于 $\odot O$ 互为反演, 称点 O 为反演中心。

性质 16.4 一条不经过反演中心的直线的反演图形是一个经过反演中心的圆。

可以根据直线到反演中心的距离与反演圆的半径的关系来互推。

性质 16.5 一个不经过反演中心的圆的反演图形还是一个不经过反演中心的圆。

可以根据圆心与反演中心的直线上与两圆的四个交点(两两对应)解出反演图形的半径和到反演中心的距离。

16.3.4.1 求过定点且与两个圆相切的圆

以定点为反演中心, 计算两个圆的反演圆, 求反演圆的公切线, 再做一次反演就可以得到经过反演中心的圆了。

以上内容参考了 ACdreamer 的博客[14]。

BZOJ2961 共点圆: 反演可以将过定点的圆周变成直线, 而直线两边的半平面就对应了圆内与圆外, 因此在所有圆中的点集就是反演半平面的交。不过圆转直线的性质还可以通过不等式化简计算, 只不过此时维护的是凸包。

16.4 半平面交

16.4.1 基本算法

基本思路是对半平面极角排序, 然后按照极角序加入半平面, 同时将无效的半平面(交点在其它半平面外)删除。这里使用双端队列实现, 同时对已加入的半平面的两个端口进行剔除。注意两半平面平行的情况, 此时保留较近的半平面。注意最后要用处理两端口 "交叉" 的情况, 应当使用一端直线消去另一端, 然后确定首尾交点作为最后一个交点。

代码如下:

[13] Smallest-circle problem - Wikipedia
https://en.wikipedia.org/wiki/Smallest-circle_problem

[14] 圆的反演变换 - ACdreamer
https://blog.csdn.net/acdreamers/article/details/16966369

```cpp
// P4196
#include <algorithm>
#include <cmath>
#include <cstdio>
int read() {
    int res = 0, c;
    bool flag = false;
    do {
        c = getchar();
        flag |= c == '-';
    } while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return flag ? -res : res;
}
typedef double FT;
const FT eps = 1e-8;
const int size = 505;
struct Vec {
    FT x, y;
    Vec() {}
    Vec(FT x, FT y) : x(x), y(y) {}
    Vec operator+(const Vec& rhs) const {
        return Vec(x + rhs.x, y + rhs.y);
    }
    Vec operator-(const Vec& rhs) const {
        return Vec(x - rhs.x, y - rhs.y);
    }
    Vec operator*(FT k) const {
        return Vec(x * k, y * k);
    }
};
FT cross(const Vec& a, const Vec& b) {
    return a.x * b.y - a.y * b.x;
}
struct Line {
    Vec ori, dir;
    FT ang;
    void init(const Vec& src, const Vec& dst) {
        ori = src;
        dir = dst - src;
        ang = atan2(dir.y, dir.x);
    }
    bool onLeft(const Vec& p) {
        return cross(dir, p - ori) > eps;
    }
    bool operator<(const Line& rhs) const {
        return ang < rhs.ang;
```

```
    }
} L[size];
Vec intersect(const Line& a, const Line& b) {
    Vec delta = a.ori - b.ori;
    FT t = cross(b.dir, delta) / cross(a.dir, b.dir);
    return a.ori + a.dir * t;
}
Vec P[size];
int q[size];
FT solve(int n) {
    std::sort(L, L + n);
    int b = 0, e = 0;
    q[0] = 0;
    for(int i = 1; i < n; ++i) {
        while(b < e && !L[i].onLeft(P[e - 1]))
            --e;
        while(b < e && !L[i].onLeft(P[b]))
            ++b;
        if(fabs(cross(L[i].dir, L[q[e]].dir)) > eps)
            q[++e] = i;
        else if(L[q[e]].onLeft(L[i].ori))
            q[e] = i;
        if(b < e)
            P[e - 1] = intersect(L[q[e - 1]], L[q[e]]);
    }
    while(b < e && !L[q[b]].onLeft(P[e - 1]))
        --e;
    if(e - b + 1 < 3)
        return 0.0;
    P[e] = intersect(L[q[b]], L[q[e]]);
    FT area = 0.0;
    for(int i = b + 1; i <= e; ++i)
        P[i] = P[i] - P[b];
    for(int i = b + 1; i < e; ++i)
        area += cross(P[i], P[i + 1]);
    return area * 0.5;
}
int main() {
    int n = read();
    int cnt = 0;
    for(int t = 0; t < n; ++t) {
        int m = read();
        for(int i = 0; i < m; ++i) {
            P[i].x = read();
            P[i].y = read();
        }
        for(int i = 0; i < m; ++i)
            L[cnt++].init(P[i],
                          P[i + 1 < m ? i + 1 : 0]);
    }
```

```
100     printf("%.3lf\n", solve(cnt));
        return 0;
}
```

一般可将线性不等式转换为半平面, 然后使用半平面交来判断是否有解/计算最优解, 除去排序后时间复杂度 $O(n)$。较复杂的最优化问题应当作为线性规划问题使用单纯形算法解决, 参见第 17.3节。

16.4.2 线性判空集

半平面交的时间复杂度为 $O(n \lg n)$, 但当只要求半平面交是否为空集时, 可以使用期望时间复杂度为 $O(n)$ 的随机化算法。

该算法维护当前半平面交的纵坐标最高点 P。每次**随机**加入新半平面 L 时, 若 P 在 L 内, 则直接跳过; 否则新的 P' 必然在 L 与之前的半平面的交点上, 用之前的半平面在当前直线上截出可行线段, 保留最高点作为新的点 P。

该方法源自 WC2012 上钟诚的讲稿《概率与随机化算法》与解轶伦的文章《随机增量算法》。

16.5 旋转卡壳

其主要思想是维护一对与多边形顶点相切的平行线(一个或两个凸多边形), 切点称为对踵点对(两个多边形时称为并踵点对)。一般当某条平行线与凸多边形的某条边相切时, 所求答案可能取得最值。维护平行线时计算其旋转到与下一条边相切的角度大小, 然后取最小角度旋转。

下面给出求对踵点对的算法:

我的方法(计算角度):算出每个向量到目标向量的角度后比较。

```
0 bool cmpAngle(const Vec& a,const Vec& b) {
      return cross(a,b)>eps;
  }
  Vec calcAngle(const Vec& src,const Vec& dst) {
      return Vec(dot(src,dst),cross(src,dst));
  }
```

为了避免 $atan2$ 精度损失, 使用 Vec 维护角, 其坐标值表示 $(k\cos\theta, k\sin\theta)$。

标准方法(枚举每个对踵点对):

```
0 void process(int i,int j);
  FT area(int a,int b,int c) {
      return cross(P[b]-P[a],P[c]-P[a]);
  }
  void solve(int n) {
      Pos i=(1,n),j(2,n);
      while(ge(area(i,i+1,j+1),area(i,i+1,j))) {
          int j0=++j;
          while(j!=n) {
              ++i;
10            process(i,j);
              while(ge(area(i,i+1,j+1),area(i,i+1,j))) {
                  ++j;
                  if(i==j0 && j==n)
```

```
                return;
            process(i,j);
        }
        if(eq(area(i,i+1,j+1),area(i,i+1,j))) {
            if(i==j0 && j==n)
                process(i+1,j);
            else
                process(i,j+1);
        }
    }
}
```

注意下标在 $[1,n]$ 内循环右移。

该算法正确性待验证, 解释留坑待补。

上述内容参考了 ACMaker 的专栏[15]。

16.6 平面最近点对

平面最近点对的经典算法采用分治思想, 即每次将点集分成等规模的两组, 递归求两组点内部的最近距离, 然后根据返回的距离来求两组间的点的最近距离。时间复杂度 $O(n \lg n)$。

步骤如下:

1. 若点集规模 ≤ 3, 直接暴力解决, 避免遇到点集规模 $= 1$ 的情况;
2. 将点分割为左右大小相等的点集(已预先按照 x 排序)递归求解;
3. 若子区间内的最小距离为 d, 则将整个区间内的距中轴距离 $\leq d$ 的点加入候选集合, 同时维持候选集合的点的纵坐标有序(可以 $O(n)$ 归并), 对于每个点与 y 值差 $\leq d$ 的点更新答案。

时间复杂度分析 关键在于证明对于每个点枚举的点数不超过某个常数。

由于左右子集都保证了其子集内部最近点对距离 $\geq d$, 使用计算步骤中所述的简单剪枝, 可以保证 $2d * d$ 的矩形中需要检查的点最多为 7 个(每边最多 4 个, 再扣除自己)。

代码如下(使用了全局最优值优化):

```
#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdio>
#include <cstdlib>
const int size = 200005;
typedef double FT;
FT read() {
    char buf[32];
    int cnt = 0, c;
    do
        c = getchar();
```

[15] https://blog.csdn.net/ACMaker/article/details/3176910
原文:http://cgm.cs.mcgill.ca/~orm/rotcal.frame.html

```
    while(!isgraph(c));
    while(isgraph(c)) {
        buf[cnt++] = c;
        c = getchar();
    }
    buf[cnt] = '\0';
    return strtod(buf, 0);
}
struct Vec {
    FT x, y;
} P[size];
bool cmpX(const Vec& a, const Vec& b) {
    return a.x < b.x;
}
bool cmpY(const Vec& a, const Vec& b) {
    return a.y < b.y;
}
FT ans2 = 1e40, ans = 1e20;
void updateDis(const Vec& a, const Vec& b) {
    FT dx = a.x - b.x, dy = a.y - b.y;
    FT dis2 = dx * dx + dy * dy;
    if(dis2 < ans2) {
        ans2 = dis2;
        ans = sqrt(dis2);
    }
}
void solve(int b, int e) {
    if(e - b <= 3) {
        for(int i = b; i < e; ++i)
            for(int j = i + 1; j < e; ++j)
                updateDis(P[i], P[j]);
        std::sort(P + b, P + e, cmpY);
    } else {
        int mid = (b + e) >> 1;
        solve(b, mid);
        solve(mid, e);
        std::inplace_merge(P + b, P + mid, P + e,
                           cmpY);
        for(int i = b; i < e; ++i) {
            FT lim = P[i].y + ans;
            for(int j = i + 1; j < e && P[j].y < lim;
                ++j)
                updateDis(P[i], P[j]);
        }
    }
}
int main() {
    int n;
    scanf("%d", &n);
    for(int i = 0; i < n; ++i) {
```

```
        P[i].x = read();
        P[i].y = read();
    }
    std::sort(P, P + n, cmpX);
    solve(0, n);
    printf("%.4lf\n", ans);
    return 0;
}
```

Chapter 17

最优化问题

17.1 最优化方法 512
17.1.1 牛顿迭代法 512
17.1.2 爬山法 513
17.1.3 模拟退火 513
17.1.4 遗传算法 514
17.1.5 A* 514
17.1.6 梯度下降 514
17.1.7 解集存储 515
17.1.8 拉格朗日乘子法 515
17.2 01 分数规划 517
17.2.1 二分答案法 517
17.2.2 Dinkelbach 法 518
17.3 线性规划 518
17.3.1 定义与规范描述形式 518
17.3.2 单纯形算法 519
17.3.3 对偶线性规划 525
17.3.4 全幺模矩阵 526
17.4 随机化算法 526
17.4.1 Monte Carlo 算法 526
17.4.2 Las Vegas 算法 527

17.1 最优化方法

下列方法主要用于解决提答题中的最优化问题~~以及乱~~搞骗~~分~~。

17.1.1 牛顿迭代法

对于简单函数尤其是多项式函数的无界最优化问题, 可以使用牛顿迭代法。首先求出待优化函数的导函数 $f(x)$, 将其转化为求 $f(x)=0$ 的根得到候选解。

选取一个初始解 x_0, 令 $x_{i+1}=x_i-\frac{f(x_i)}{f'(x_i)}$, 迭代几次后就会收敛。

$f(x)$ 最好在整个定义域内二阶可导, 由于 $f(x)$ 有多个根, 需要用多个初始解 x_0 得到不同的根。

马同学高等数学[1]中提到, 有些特殊函数的根并不能使用牛顿迭代法求解。

17.1.2 爬山法

每次迭代时选取与 x 点临近的点 x' 计算, 若其值更优, 则接受 x'(也可以选择临近最优点), 进入下一步迭代; 若所有临近点都比 x 要差, 那么此时 x 局部最优。重新随机选取初始点进行迭代, 将迭代过的点中最优解作为答案。

17.1.2.1 随机爬山法

在选取下一迭代点时, 还可以对更优点集按照变化值建立分布, 然后离散随机采样。

17.1.3 模拟退火

模拟退火为比赛中的常用方法。步骤如下:

1. 选取一个初始温度 T, 与冷却系数 $d \in (0,1)$, 初始化一个解 s, 计算 $f(s)$;
2. 稍微修改该解得到新解 s', 计算 $f(s')$;
3. - 如果 $f(s') > f(s)$, 则直接接受新解 s';
   - 否则按照 Metropolis 准则, 以 $e^{-\frac{\Delta f}{T}}$ 的概率接受新解。
4. 冷却, 即 $T \leftarrow Td$;
5. 当 T 足够小时退出, 否则返回步骤 2。

T 与 d 的作用是让变化越来越 "保守"。在整个模拟退火的过程中要另外维护一个搜索到的最优解, 以增加正确率。

在得到解后为了避免舍入误差, 可以在最优解周边随机一些点更新。如果是二维选点问题, 可以将局部区域当做凸函数, 然后三分套三分解决。

17.1.3.1 参数选取

参数 T 与 d 的选取是玄学。。。可以结合从某解到最优解的最大/期望/最小步数考虑。

Update: 在 FlashHu 的博客里找到了人肉调参方法[2]: 调参时二分初始温度与冷却系数(主要调冷却系数), 每个二分值测试多次(因为随机化)。迭代输出当前解与对应温度, 用心感受解的下降速度是否均匀, 如果下降速度太快或太慢就调整对应时间段内的冷却系数。或许有自动化的算法完成这一过程? 启发式模拟退火?

17.1.3.2 新解生成

注意在计算新解的价值时, 尽量由旧解的价值修改得出, 以降低更新解的复杂度。在变化时要尽量避免价值差变化过大(比如 TSP 中交换相邻城市比交换任意两个城市更好)。当然为了防止解在一个 "盆地" 中过分逗留, 可以重新选取初始解或者选择恰当时机对当前解进行比较大的改变。随着温度的减小, 对解的变动也应该减小。

---

[1]如何通俗易懂地讲解牛顿迭代法? https://www.matongxue.com/madocs/205/

[2]模拟退火总结(模拟退火)
https://www.cnblogs.com/flashhu/p/8884132.html

17.1.3.3 分块模拟退火

对于多峰函数, 将区间分成几块, 在块内使用模拟退火(因为模拟退火利用相邻解的关系)。

以上内容参考了 Wikipeida-EN[3]

17.1.4 遗传算法

遗传算法引入了生物学中的概念:遗传、变异、竞争与淘汰。

算法步骤如下:

1. 生成一些初始解, 计算这些解的 ‘‘适应度’’;
2. 选取这些解的一部分较优解;
3. 通过较优解之间的配对交叉把自己的 ‘‘基因’’(编码)遗传给子代(可以加权), 子代的基因以一定的概率变异(只要考虑 $p < 0.5$), 加入下一代群体;
4. 迭代固定次数后退出, 否则返回步骤 2;
5. 输出计算历史中适应度最高的解。

17.1.5 A*

A* 算法通过引入一个估价函数 $h(x)$ 表示到目标点长度的估计值, 在做最短路时使用 $f(x) + h(x)$ 作为权重更新距离, 可以达到比 Dijkstra 更好的性能。

17.1.5.1 IDA*

IDA* 比 A* 多了一个迭代加深的操作, 即每次 DFS/BFS 搜索时限制搜索深度。若使用当前深度参数无法找到可行解, 就加深深度重新 DFS, 注意这里不一定要保存上一次的搜索树, 因为新搜索树比原搜索树大得多。

17.1.6 梯度下降

梯度下降法用来找局部最优值, 其主要思想是利用当前点的梯度来引导寻找更优值。梯度在笛卡尔坐标系中被定义为一个向量, 其坐标值为目标函数在对应基向量上的偏导数。若要寻找 $f(x)$ 最小值, 迭代时令 $x_{n+1} = x_n - \gamma \nabla f(x_n)$。

注意 γ 要小且不需要更改, 因为随着算法的收敛, $\nabla f(x)$ 会越来越小。迭代时达到指定精度或者固定迭代次数后直接退出, 注意要不断更换 γ 来保证算法的收敛(γ 过大会导致波动幅度大, 过小会导致收敛速度慢)。

Barzilai-Borwein Method 这是一种自适应设置步长 γ 的方法, 其表达式如下:

$$\begin{aligned} D &= \nabla f(x_n) - \nabla f(x_{n-1}) \\ \gamma_n &= (x_n - x_{n-1})^T \cdot \hat{D} \end{aligned}$$

$\hat{D}$ 表示 D 对应的单位向量。

例如求 $f(x) = x^4 - 3x^3 + 2$ 的在 $x_0 = 6$ 附近的最小值所对应的 x:

[3]Simulated annealing - Wikipedia
https://en.wikipedia.org/wiki/Simulated_annealing

```cpp
#include <cmath>
#include <cstdio>
typedef double FT;
const FT p = 1e-12, g = 1e-3;
const int maxStep = 1 << 20;
FT f(FT x) {
    return (x - 3.0) * x * x * x + 2.0;
}
FT df(FT x) {
    return (4.0 * x - 9.0) * x * x;
}
int main() {
    FT cur = 6.0;
    int step;
    for(step = 0; step < maxStep; ++step) {
        FT delta = -g * df(cur);
        cur += delta;
        if(fabs(delta) < p)
            break;
    }
    printf("step=%d res=(%.12lf,%.12lf)\n", step, cur,
           f(cur));
    return 0;
}
```

上述内容参考了 Wikipedia-EN[4]。

17.1.7 解集存储

在解决最优化问题时, 保存新解**当且仅当该解比存储的最优解更优且该解即将被比自己更差的解替代作为当前解**, 在迭代退出再执行一遍该逻辑。此法延迟了解集保存时间, 可以减少冗余拷贝。Lazy Evaluation 大法好!!!

17.1.8 拉格朗日乘子法

拉格朗日乘子法用于求解**多元函数条件极值问题**。给定两个一阶连续可导函数 $f(X), g(X)$, X 为变量组, 在满足 $g(X) = 0$ 的条件下最优化 $f(X)$。也存在 $g(X) = c$ 的形式, 常数一般不隐藏在 $g(X)$ 中。

拉格朗日乘子法引入了一个新的变量 λ, 称作拉格朗日乘子。然后构造一个新的函数 $\mathcal{L}(X, \lambda) = f(X) - \lambda g(X)$, 称作拉格朗日函数。

我们在 $g(X) = 0$ 的区域上移动, 如果在某处 $f(X)$ 局部不改变的话, 这个地方很有可能取得极值, 且极值点只可能在这些点取到。考虑 "等势区域"$g(X) = 0$ 与 $f(X) = d$, $f(X)$ 的值不改变意味着它也在等势区域上移动。那么有这两个等势区域在该处平行, 意味着这个 X 在 $f(X)$ 与 $g(X)$ 上的梯度平行, 记为 $\nabla_X f = \lambda \nabla_X g$。再加上约束 $g(X) = 0$, 结合先前引入的拉格朗日函数, 有 $\nabla_{X,\lambda} \mathcal{L} = 0$。

在实际计算中, 首先根据 $\nabla_X \mathcal{L} = 0$ 用 λ 表达出 X, 然后代入 $g(X) = 0$ 求出可行的 λ。这时解的搜索范围被极大地缩小, 直接比较每个 λ 对应的 $f(X)$, 得到所需极值。多个 $g(X) = 0$ 的处理方法类似。

[4]Gradient descent - Wikipedia https://en.wikipedia.org/wiki/Gradient_descent

上述内容参考了 Wikipedia-EN[5]。

例题 「NOI2012」骑行川藏

很容易将其转化为多元函数条件极值问题：

- 满足约束 $g(V)=\sum k_i(v_i-v_i')^2 s_i = E_U$
- 最小化 $f(V)=\sum \frac{s_i}{v_i}$

求偏导得 $\nabla_{v_i}\mathcal{L}=-s_i v_i^{-2}-2\lambda s_i k_i(v_i-v_i')$，令其为 0，化简得 $2\lambda k_i(v_i-v_i')v_i^2+1=0$。注意到 λ 为负数，λ 越大，每个 v_i 越大，耗费的能量 $g(V)$ 越大。

那么可以二分 λ，根据式子使用牛顿迭代法求出 v_i，回代求出 $g(V)$，与 E_U 比较。

一个比较 trick 的方法是以上一次迭代的解为初始解，当 $\delta<\varepsilon$ 时退出迭代，得到不错的性能提升。~~对着数据调参数~~

参考代码：

```
#include <cmath>
#include <cstdio>
typedef double FT;
const int size = 10005, IA = 15, IB = 70;
FT s[size], k[size], sk[size], vf[size], v[size];
inline FT f(int i, FT lambda, FT v) {
    return 2.0 * lambda * k[i] * (v - vf[i]) * v * v +
        1.0;
}
inline FT df(int i, FT lambda, FT v) {
    return 2.0 * lambda * k[i] *
        (3.0 * v - 2.0 * vf[i]) * v;
}
const FT eps = 1e-13;
void getV(int n, FT lambda) {
    for(int i = 1; i <= n; ++i) {
        for(int j = 0; j < IA; ++j) {
            FT delta = f(i, lambda, v[i]) /
                df(i, lambda, v[i]);
            if(fabs(delta) < eps)
                break;
            v[i] -= delta;
        }
    }
}
int main() {
    int n;
    FT EU;
    scanf("%d%lf", &n, &EU);
    for(int i = 1; i <= n; ++i) {
        scanf("%lf%lf%lf", &s[i], &k[i], &vf[i]);
        sk[i] = s[i] * k[i];
```

[5]Lagrange multiplier
https://en.wikipedia.org/wiki/Lagrange_multiplier

```
            v[i] = 200.0;
        }
        FT l = -1.0, r = 0.0;
        for(int i = 0; i < IB; ++i) {
            FT m = (l + r) * 0.5;
            getV(n, m);
            FT E = 0.0;
            for(int j = 1; j <= n; ++j) {
                FT dv = v[j] - vf[j];
                E += sk[j] * dv * dv;
            }
            (E < EU ? l : r) = m;
        }
        getV(n, l);
        FT t = 0.0;
        for(int i = 1; i <= n; ++i)
            t += s[i] / v[i];
        printf("%.8lf\n", t);
        return 0;
}
```

17.1.8.1 KKT 条件

KKT 条件是拉格朗日乘子法的扩展, 用于求解不等式约束下的最优化问题。
看上去不太好求解, 留坑待补。
更好的理解方式参见马同学高等数学[6]。

17.2 01 分数规划

该问题可表述为有一堆物品, 每个物品有价值 a_i 与代价 b_i, 询问选择的物品集合所能得到的 $\frac{\sum_{i\in S} a_i}{\sum_{i\in S} b_i}$ 的最大值。

注意当 $\sum_{i\in S} a_i$ 或 $\sum_{i\in S} b_i$ 表示为 $|S|$ 时, 这是一个隐式的分数规划问题。血泪史: [SCOI2014] 方伯伯运椰子

17.2.1 二分答案法

设答案为 x, 存在点集 S 满足 $\frac{\sum_{i\in S} a_i}{\sum_{i\in S} b_i} \geq x$, 将该式转化为 $\sum_{i\in S} a_i - x \cdot \sum_{i\in S} b_i \geq 0$。因此可二分答案 x, 对原数据进行一些修改, 便可将原问题转化为解决一个判定问题(或是新的

[6]如何理解拉格朗日乘子法? https://www.matongxue.com/madocs/939/
如何理解拉格朗日乘子法和 KKT 条件? https://www.matongxue.com/madocs/987/

最优化问题, 即检查 $f_x(S) = \sum_{i\in S} a_i - x \cdot \sum_{i\in S} b_i$ 的最大值是否不小于 0)。

一般使用网络流/SPFA 判负环/MST 辅助判定。

例如对于最小平均值环问题, 二分环平均长度 x, 将每条边的长度减去 x, 然后判断负环, 得到下一次迭代的二分范围。

17.2.2 Dinkelbach 法

记选择方案 S 对应直线 $f_S(x) = \sum_{i\in S} a_i - x \cdot \sum_{i\in S} b_i$, 该直线的横截距为该方案的目标函数值。若 $f_S(x) \geq 0$ 则说明该直线所对应的横截距 $\geq x$。

在二分过程中检查二分点 m 的合法性时, 有时通过求 $max\{f_m(S)\}$ 来判定。满足检查条件时仅仅令 $L = m$ 就显得浪费了, 因为检查下一二分点时得到的最优解 S 可能不变。Dinkelbach 法的思路是维护答案的下界, 使用当前最优解对应的直线横截距来当做下一次迭代的起点, 这种做法比二分法更快。不过需要保存解这一要求限制了它的应用范围(编码更麻烦)。

事实上解不必逐步递增, 也可以使用类似牛顿迭代法的思想, 解可以左右移动, 但是每次迭代要选取最大的移动截距(例如取最大值时尽可能向右移, 尽可能少后退)。选取一个好的初始解可以大幅加快迭代速度。

此法参考了 tianxiang971016 的博客[7]。

上述方法的性能比较(以 LuoguP4292 [WC2010] 重建计划的评测结果为准):

- 二分法:9622ms
- Dinkelbach 法:1864ms
- Dinkelbach 法优化:1085ms

~~2019.3.3:为什么又是 rank2。。。~~

当无法求得最大值时, 二分法比 Dinkelbach 法效率更高。而且 Dinkelbach 法可能求出值恰好为 0 的解导致提前收敛, 结果错误。这时可以给截距加上一个扰动, 令其越过收敛点, 如果后续的解可行则继续迭代, 即使不可行扰动也不会影响答案。

17.3 线性规划

17.3.1 定义与规范描述形式

17.3.1.1 定义

线性函数 给定 n 个实数 $a_1, a_2, \cdots, a_n$ 与 n 个变量 $x_1, x_2, \cdots, x_n$, 线性函数 f 是这些变量的线性加权和, 即

$$f(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{n} a_i x_i$$

线性约束 给定一个实数 b, 线性约束是满足 $f(x_1, x_2, \cdots, x_n) = b, \leq b$ 或 $\geq b$ 的线性等式/不等式。

线性规划 一个线性规划问题是在使一组变量满足一组有限个线性约束的前提下, 最大(小)化某个线性函数值。

[7] 01 分数规划问题相关算法与题目讲解(二分法与 Dinkelbach 算法)- ztx
https://blog.csdn.net/hzoi_ztx/article/details/54898323

可行解 满足所有线性约束的解。

目标函数与目标值 目标函数是我们希望最大(小)化其值的线性函数, 目标值是特定变量组合对应的目标函数值。

线性规划的解 一个线性规划称为不可行的当且仅当它没有可行解。一个可行线性规划称为无界的当且仅当它没有最优值。

矩阵表示 下面的内容中, 大小为 n 的向量 x 表示由变量 $x_1, x_2, \cdots, x_n$ 组成的向量, 大小为 n 的向量 c 表示目标函数 $f : x \to c^T x$ 的系数, 大小为 $m * n$ 的矩阵 A 表示 m 个约束的系数, 大小为 n 的向量 b 表示由这 m 个约束的常数项组成的向量。

17.3.1.2 标准型

标准型的描述如下:

- 最大化 $c^T x$
- 满足约束 $Ax \leq b$
- 满足非负约束 $x \geq 0$

任意线性规划都可以按照如下方法将其转换为等价的标准型线性规划:

- 要求最小化目标函数值:将目标函数的系数取反。
- 变量不具有非负约束:若变量 x_i 没有非负约束, 则引入两个非负变量 x_{i1}, x_{i2}, 满足 $x_i = x_{i1} - x_{i2}$。然后把线性规划中的 x_i 替换为 $x_{i1} - x_{i2}$。
- 约束中有等式约束:将 $f(x) = b$ 拆为 $f(x) \geq b$ 和 $f(x) \leq b$。
- 约束中有 $f(x) \geq b$ 形式的约束:约束的系数与常数均取反。

17.3.1.3 松弛型

单纯形算法需要把不等式约束转换为等式约束。对于每一个约束 $f_i(x) \leq b_i$, 引入一个非负**松弛变量**x_{n+i} 使得 $x_{n+i} = b_i - f_i(x)$。记目标函数值为 z, 则也可以引入等式 $z = c^T x$。在此称等式左边的变量为**基本变量**, 等式右边的变量称为**非基本变量**。实际上基本变量与目标函数值都被表示为常数 + 非基本变量线性加权和的形式。等式组的变量不同时出现于等号两边, 基本变量也不会出现两次。**注意等式左边的变量不等同于松弛变量, 因为单纯形算法的转动操作会使变量的位置改变。**

17.3.2 单纯形算法

17.3.2.1 原理

基本解 将线性规划问题转换为松弛型, 令非基本变量的值为 0, 就可以确定基本变量的值与目标函数值。这是该线性规划的一个**基本解**。若它对应的基本变量的值均非负, 则说明它是一个**基本可行解**。

单纯形算法的步骤就是:

- 找到初始基本可行解, 若没有则说明该线性规划不可行。
- 不断迭代:计算当前基本解对应的解和目标函数值, 判断是否最优。如果不是, 根据计算结果执行 ‘‘转动’’ 操作更换基本变量得到更优解。

17.3.2.2 转动 pivot

转动过程每次选取一个非基本变量 x_e 和一个基本变量 x_l, 交换它们在约束中的位置。因此 x_e 称为替入变量, x_l 称为替出变量。

在实现中用 $A[m][n]$ 表示约束和目标函数, $id[1\cdots n]$ 表示等式右边每列的非基本变量编号, $id[n+1\cdots n+m]$ 表示等式左边每列的基本变量编号。行 $A[0]$ 用于存储目标函数信息。$A[i][0]$ 满足

$$A[i][0] + \text{目标函数值} z = \sum_{j=1}^{n} A[i][j]x_{id[j]} \text{ if } i = 0$$

该式可由 $z = c^T x$ 推导。

或

$$A[i][0] - x_{id[n+i]} = \sum_{j=1}^{n} A[i][j]x_{id[j]} \text{ otherwise}$$

该式可由 $x' = b - f(x)$ 推导。

令非基本变量值为 0 后, $-A[0][0]$ 就是目标函数值 z, $A[i][0](i = 1\cdots m)$ 就是 $x_{id[n+i]}$ 的值。初始化矩阵时, 令 $A[0][0]$ 为 0, 其余参数直接填入对应位置, 然后让 $id[i] = i(i = 1\cdots n)$。由于最终我们只要 $x_i(i = 1\cdots n)$ 的值, 不必初始化松弛变量的编号。

执行转动时, 首先交换对应位置的变量编号, 然后处理替出变量所在约束的矩阵行, 最后处理其余行。

处理替出变量所在行 记替入变量为 $x_{id[e]}$, 替出变量为 $x_{id[n+l]}$, 该行 $L = A[l]$ 除 $x_{id[e]}, x_{id[n+l]}$ 外的其余非基本变量组成的向量为 X, 系数向量为 c, 有

$$\begin{aligned} L[0] - x_{id[n+l]} &= c^T X + L[e]x_{id[e]} \\ \Rightarrow \frac{L[0]}{L[e]} - \frac{1}{L[e]}x_{id[n+l]} &= \frac{c^T}{L[e]}X + x_{id[e]} \\ \Rightarrow \frac{L[0]}{L[e]} - x_{id[e]} &= \frac{c^T}{L[e]}X + \frac{1}{L[e]}x_{id[n+l]} \end{aligned}$$

处理其余行 类似于高斯消元法, 用行 $A[l]$ 消去其它行中的 $A[i][e]$ 项。注意还要计算替出变量的系数, 所以 $A[i][e]$ 要先置 0。

性质 如果当前基本解是可行解且 pivot 操作的参数 l, e 满足 $A[l][0]/A[l][e]$ 是所有满足 $A[l][e] > 0$ 的行中的最小值, 那么 pivot 后得到的基本解仍然是一个可行解。

证明:

- 替出变量所在行: 由于 $L[0]$ 非负且 $L[e]$ 为正, pivot 后 $x_{id[e]}$ 的值 $\frac{L[0]}{L[e]}$ 非负。
- 其余行: 由于 $A[l][0]/A[l][e]$ 最小, 转动替出变量所在行后该行的常数项 $A[l][0]$ 最小。设当前要消第 j 行:
  - 若 $A[j][e] > 0$, 则满足 $A[l][0]/A[l][e] \le A[j][0]/A[j][e]$。变换得 $A[j][0]' = A[j][0] - A[l][0]/A[l][e] * A[j][e] \ge 0$, 仍然保持非负。
  - 若 $A[j][e] \le 0$, 则 $A[j][0]$ 消元后不减, 仍然保持非负。

17.3.2.3 主过程 simplex

判断是否为最优解 当对固定的基本变量组进行矩阵行变换后目标函数中所有的系数都非正, 则说明当前基本解是最优解。

证明: 由于目标函数值只取决于非基本变量的值, 且基本解中非基本变量均为 0, 只能提升非基本变量的值来改变目标函数值。由于系数均非正, 即使提升非基本变量的值仍然可行, 目标函数值也不增。

选取替入/替出变量 选取目标函数值中系数非正的非基本变量进行提升是无用的, 因此要选取一个满足 $A[0][e] > 0$ 的变量当做替入变量。由 pivot 的性质可得只有 $A[l][e] > 0$ 且 $A[l][0]/A[l][e] > 0$ 才能在 pivot 操作后保证其仍然为基本可行解。也就是说, 一旦确立了替入变量, 替出变量的选择就被限制了。在使用 pivot 操作提升非基本变量后, 不仅保证了当前基本解仍然是可行解, 还能提升目标函数值(除非 pivot 后替入变量的值为 0)。目标函数值不提升的现象被称为**退化**, 可能导致算法出现循环无法终止, 因此需要使用 Bland 规则来避免循环。

根据 Bland 规则, 总是选取下标最小的替入变量以及对应最优且下标最小的替出变量, 可以避免单纯形算法的循环。使用 Bland 规则后, 可以保证算法在 $\binom{n+m}{m}$ 次迭代内终止, 当然实践中单纯形算法的表现很好, 很少遇到如此刁钻的数据(或许随机化有助于改善算法运行时间)。

算法每次迭代后的目标函数值不降, 且算法会在有限次迭代内结束(枚举完所有可能的基本变量组后), 因此算法会输出最优解。**这里还需要证明基本可行解集合中含有最优解。不过基本解对应了凸集中的边界, 根据经验最优解一定在边界上取得。更标准的证明留坑待补。**

判断无界 若存在系数非负的候选替入变量但不存在对应的替出变量, 则该线性规划是无界的。

不存在对应的替出变量意味着所有 $A[i][e] \le 0$, 那么任意提高该替入变量的值, 仍然保持其它非基本变量为 0, 既能保持当前基本解仍然为可行解, 还能提高目标函数值。因此该线性规划是无界的。

17.3.2.4 初始化 init

找到一个初始基本可行解意味着要让松弛型中约束的常数项非负。

辅助线性规划 首先检查初始松弛型是否已经对应了基本可行解。记 k 为满足 $A[k][0]$ 最小的下标, 若基本解不可行则有 $A[k][0] < 0$。

然后引入一个新变量 x_0, 和原有的线性规划组成一个新的标准型线性规划:

- 最大化 $-x_0$
- 满足约束 $Ax - x_0 \le b$
- 满足约束 $x, x_0 \ge 0$

原线性规划可行当且仅当该线性规划的最优值是 0。

首先构造出该线性规划 B, 其中 x_0 放置于非基本变量组的末尾。然后执行 $pivot(k, n_B)$, 这样就得到了线性规划 B 的一个基本可行解。

证明:

- 替出变量所在行: $L[0]$ 为负且 $L[e] = -1$, 转动后 $L[0]' = -L[0] > 0$。
- 其余行: 由于 $A[i][e] = -1$, 每行都要加上一倍 L, 由于 $L[0]$ 是所有 $A[i][0]$ 中的最小项, 所以 $A[i][0]' \ge 0$。

然后对 B 运行 simplex 找到最优值, 最优值不为 0 则返回不可行。

若 x_0 此时为基本变量, 在其对应行中选取一个系数非 0 的非基本变量作为替入变量, 把 x_0 换出去。

注意:

- pivot 操作并不会使最优解变小, 因为转动后由基本解的定义可知 $x_0 = 0$。
- pivot 操作不会让基本解不可行, 因为 $L[0] = 0$, 其它的 $A[i][0]$ 仍然保持非负。

移除 x_0, 把 B 的结果写回原线性规划中, 完成初始化。

随机初始化 不断迭代, 每次随机选一个 l 满足 $A[l][0] < 0$, 再随机选择一个 e 满足 $A[l][e] < 0$, 执行 pivot 后可以使 $A[l][0]$ 为正。

避免初始化 可以根据题目性质, 松弛一些约束(比如将 $=$ 松弛为 $\leq$), 使得初始线性规划就是合法的松弛形。

17.3.2.5 算法实现(UOJ179)

为了过掉 UOJ 的 Extra Test, 这里使用了两种初始化方法的混合(然而还是过不去, 只能指望 Mehrotra predictor–corrector method 了)。

Update:事实上应该是浮点数精度不够。

```
#include <algorithm>
#include <cmath>
#include <iostream>
typedef long double FT;
const FT eps = 1e-10;
const int size = 25, maxStep = 2000;
struct LP {
    FT A[size][size];
    int id[size * 2], n, m;
    void pivot(int l, int e) {
        std::swap(id[n + l], id[e]);
        FT fac = A[l][e];
        A[l][e] = 1.0;
        for(int i = 0; i <= n; ++i)
            A[l][i] /= fac;
        for(int i = 0; i <= m; ++i)
            if(i != l) {
                FT k = A[i][e];
                A[i][e] = 0.0;
                for(int j = 0; j <= n; ++j)
                    A[i][j] -= k * A[l][j];
            }
    }
    bool simplex(int m) {
        while(true) {
            int e = 0;
            for(int i = 1; i <= n; ++i)
                if(A[0][i] > eps) {
```

```
                    e = i;
                    break;
                }
            if(e == 0)
                break;
            FT minv = 1e20;
            int l = 0;
            for(int i = 1; i <= m; ++i)
                if(A[i][e] > eps) {
                    FT val = A[i][0] / A[i][e];
                    if(val < minv) {
                        minv = val;
                        l = i;
                    }
                }
            if(l == 0)
                return false;
            pivot(l, e);
        }
        return true;
    }
} A, B;
int getRandom() {
    static int seed = 2354;
    return seed = seed * 48271LL % 2147483647;
}
int q[size];
int initRandom() {
    int res = 0;
    for(int i = 0; i < maxStep; ++i) {
        int qcnt = 0;
        for(int i = 1; i <= A.m; ++i)
            if(A.A[i][0] < -eps)
                q[qcnt++] = i;
        if(qcnt == 0) {
            res = 1;
            break;
        }
        int l = q[getRandom() % qcnt];
        qcnt = 0;
        for(int i = 1; i <= A.n; ++i)
            if(A.A[l][i] < -eps)
                q[qcnt++] = i;
        if(qcnt == 0)
            return -1;
        int e = q[getRandom() % qcnt];
        A.pivot(l, e);
    }
    return res;
}
```

```
bool init() {
    int res = initRandom();
    if(res != 0)
        return res == 1;
    int k = 1;
    for(int i = 2; i <= A.m; ++i)
        if(A.A[i][0] < A.A[k][0])
            k = i;
    if(A.A[k][0] > -eps)
        return true;
    B = A;
    ++B.n, ++B.m;
    B.id[B.n] = B.n;
    for(int i = 1; i < B.n; ++i)
        B.A[0][i] = 0.0;
    for(int i = 0; i <= B.m; ++i)
        B.A[i][B.n] = -1.0;
    for(int i = 0; i <= A.n; ++i)
        B.A[B.m][i] = A.A[0][i];
    B.pivot(k, B.n);
    B.simplex(B.m - 1);
    FT ans = -B.A[0][0];
    if(ans < -eps)
        return false;
    for(int i = 1; i <= B.m; ++i)
        if(B.id[B.n + i] == B.n) {
            for(int j = 1; j <= B.n; ++j)
                if(fabsl(B.A[i][j]) > eps) {
                    B.pivot(i, j);
                    break;
                }
            break;
        }
    for(int i = 1; i <= B.n; ++i)
        if(B.id[i] == B.n) {
            for(int j = 1; j < i; ++j)
                A.id[j] = B.id[j];
            for(int j = i + 1; j <= B.n; ++j)
                A.id[j - 1] = B.id[j];
            for(int j = 1; j <= A.m; ++j)
                A.id[j + A.n] = B.id[j + B.n];
            for(int j = 1; j <= B.m; ++j) {
                int nj = j == B.m ? 0 : j;
                for(int k = 0; k < i; ++k)
                    A.A[nj][k] = B.A[j][k];
                for(int k = i + 1; k <= B.n; ++k)
                    A.A[nj][k - 1] = B.A[j][k];
            }
            break;
        }
```

```
    return true;
}
bool simplex() {
    if(!init()) {
        std::cout << "Infeasible" << std::endl;
        return false;
    }
    if(!A.simplex(A.m)) {
        std::cout << "Unbounded" << std::endl;
        return false;
    }
    return true;
}
FT ans[size];
int main() {
    int t;
    std::cin >> A.n >> A.m >> t;
    for(int i = 1; i <= A.n; ++i)
        std::cin >> A.A[0][i];
    for(int i = 1; i <= A.m; ++i) {
        for(int j = 1; j <= A.n; ++j)
            std::cin >> A.A[i][j];
        std::cin >> A.A[i][0];
    }
    for(int i = 1; i <= A.n; ++i)
        A.id[i] = i;
    std::cout.precision(14);
    if(simplex()) {
        std::cout << std::fixed << -A.A[0][0]
                  << std::endl;
        if(t) {
            for(int i = 1; i <= A.m; ++i)
                ans[A.id[A.n + i]] = A.A[i][0];
            for(int i = 1; i <= A.n; ++i)
                std::cout << std::fixed << ans[i]
                          << ' ';
        }
    }
    return 0;
}
```

17.3.2.6 稀疏矩阵优化

单纯形算法性能的关键在于 pivot 操作。如果矩阵 A 为稀疏矩阵, 则首先考虑使用其它算法解决。一定要用单纯形算法解决的, 可以扫描一遍行 L 记录非零元素到队列, 消元时也判断一下系数是否为 0。

17.3.3 对偶线性规划

标准型的对偶线性规划为:

- 最小化 $b^T y$
- 满足约束 $A^T y \geq c$
- $y \geq 0$

标准型和它的对偶线性规划最优值相等, 即 $c^T x = b^T y$。
对偶线性规划用来转化问题, 但要慎用单纯形算法计算。
要注意转化问题时要不要加入选择个数限制, 有时由于贪心的缘故可以去掉这个限制以简化问题。

17.3.4 全幺模矩阵

一个矩阵 A 是**全幺模矩阵**的充分条件如下:

- A 的元素仅有-1, 0, 1。
- 每列最多有 2 个非 0 数。
- 行可以分为 2 个集合, 根据列来划分集合:
  - 若列中有 2 个同号非 0 数, 两行不在同一集合
  - 若列中有 2 个异号非 0 数, 两行在同一集合

若线性规划的系数矩阵 A 为全幺模矩阵, 或许可以用 int 代替 double 存储矩阵。
任何最大流以及最小费用最大流问题的线性规划矩阵都是全幺模矩阵。~~如果发现线性规划矩阵是个全幺模矩阵, 可以考虑将其转化为简单的网络流模型来做~~。
上述内容参考了算法导论 [4] 第 29 章与 Angel_Kitty 的博客[8], 该篇文章末尾引用了 Candy? 的博客[9]。网上博客的术语定义杂乱, 这里使用算法导论中的定义。

17.4 随机化算法

随机化算法一般用来解决判定性问题与最优化问题。

17.4.1 Monte Carlo 算法

17.4.1.1 有关枚举元素的问题

该问题需要枚举每个子集, 然后对单个枚举进行计算。
对于这种问题我们有复杂度无法接受的穷举法, 将穷举改为随机选取固定次数的子集, 可以在规定时间内完成计算。
使用一些贪心技巧或利用题目性质可以提高单次枚举正确率与采样数。

17.4.1.2 线性加权和问题

每次随机生成一个权重向量与向量做内积/与矩阵做乘法。

17.4.1.3 复杂度与正确率

OI 中一般只需设计产生单侧错误算法。
若单次正确率为 p, 复杂度为 $O(f(n))$, 则测量 k 次的正确率为 $1-(1-p)^k$, 复杂度为 $O(kf(n))$。

[8]线性规划之单纯形法【超详解 + 图解】http://www.cnblogs.com/ECJTUACM-873284962/p/7097864.html
[9][单纯形法与线性规划]【学习笔记】https://www.cnblogs.com/candy99/p/lp.html

17.4.2 Las Vegas 算法

留坑待补。

上述内容参考了国家集训队 2014 论文集胡泽聪的《随机化算法在信息学竞赛中的应用》。

Chapter 18

理论

18.1 时间复杂度分析 528
18.1.1 主定理 528
18.1.2 Akra-Bazzi 法 529
18.2 数值编码 529
18.2.1 整数编码 529
18.2.2 浮点数编码 530
18.3 常见排序算法 530
18.3.1 比较排序算法 530
18.3.2 非比较排序算法 530
18.4 NP 完全性 530
18.4.1 定义 530
18.4.2 常见 NPC 问题 531

18.1 时间复杂度分析

18.1.1 主定理

定理 18.1 (Master Theorem) 对于递归式

$$T(n) = aT(n/b) + f(n)$$

有如下渐近界：

- 若对常数 $\varepsilon > 0$，有 $f(n) = O(n^{\log_b a - \varepsilon})$，则 $T(n) = \Theta(n^{\log_b a})$。
- 若 $f(n) = \Theta(n^{\log_b a})$，则 $T(n) = \Theta(n^{\log_b a} \lg n)$
- 若对常数 $\varepsilon > 0$，有 $f(n) = \Omega(n^{\log_b a + \varepsilon})$，则 $T(n) = \Theta(f(n))$。

简单来说，先比较函数 $f(n)$ 和 $n^{\log_b a}$ 的渐近大小，若不同则选择较大的一个，相同则再乘个 $\lg n$。

证明留坑待补。

18.1.1.1 特例

定理 18.2 若 $f(n)=\Theta(n^{\log_b a}\lg^k n)$,其中 $k\geq 0$,则主递归式的解为 $T(n)=\Theta(n^{\log_b a}\lg^{k+1} n)$。

证明:注意此例不能使用主定理,考虑对递归式进行展开,假设 n 为 b 的幂,有

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_b n} a^i\Theta(n^{\log_b a}\lg^k n)\\
&= \Theta\left(\sum_{i=0}^{\log_b n}\left(a^i\left(\frac{n}{b^i}\right)^{\log_b a}(\lg n-i)^k\right)\right)\\
&= \Theta\left(\sum_{i=0}^{\log_b n}\left(n^{\log_b a}(\lg n-i)^k\right)\right)\\
&= \Theta\left(n^{\log_b a}\sum_{i=0}^{\log_b a}(\lg n-i)^k\right) \text{ 因为}\lg n\leq\log_b a\\
&= \Theta\left(n^{\log_b a}\lg^{k+1} n\right)
\end{aligned}$$

以上内容参考了算法导论 [4] 第 4.5 节。

18.1.2 Akra-Bazzi 法

对于子问题规模划分不均衡的算法,不能使用主方法,Akra-Bazzi 法解决了如下递归式的渐进界计算:

$$T(x)=\begin{cases}\Theta(1) & 1\leq x\leq x_0\\ \sum_{i=1}^k a_iT(b_ix)+f(x) & x>x_0\end{cases}$$

其中常数 $x_0\geq 1/b_i$ 且 $x_0\geq 1/(1-b_i), 0<b_i<1, a_i>0$, $f(x)$ 满足存在正常数 c_1,c_2 使得对于 $b_ix\leq u\leq x$,有 $c_1f(x)\leq f(u)\leq c_2f(x)$(或者 $|f'(x)|$ 的上界为多项式时也满足条件)。

首先计算满足 $\sum_{i=1}^k a_ib_i^p=1$ 的 p,然后可得

$$T(n)=\Theta\left(x^p\left(1+\int_1^x\frac{f(u)}{u^{p+1}}\mathrm{d}u\right)\right)$$

上述方法参考了算法导论 [4] 第 4 章的本章注记。

18.2 数值编码

18.2.1 整数编码

以下三种编码的首位都是符号位,正 0 负 1。

原码 数值位为真值的绝对值。

反码 正数不变,负数数值位为原码数值位取反。

补码 正数不变, 负数数值位为原码数值位取反 +1。补码转反码时-1 取反或者取反 +1。

18.2.2 浮点数编码

留坑待补。

18.3 常见排序算法

18.3.1 比较排序算法

算法	复杂度(最好/平均/最坏)	稳定性	原址排序
快速排序	$n\lg n/n\lg n/n^2$		√
归并排序	$n\lg n$	√	
堆排序	$n\lg n$		√
插入排序	$n/n^2/n^2$	√	√
选择排序	n^2		√
希尔排序	$n\lg n/-/n^{4/3}$		√
冒泡排序	$n/n^2/n^2$	√	√

18.3.2 非比较排序算法

- 计数排序
- 基数排序
- 桶排序

上述内容参考了 Wikipedia-EN[1]

18.4 NP 完全性

18.4.1 定义

P P 类问题是在多项式时间内可被解决的问题。

NP NP 类问题是在多项式时间内解可被检验的问题。

NP-Complete NP 完全问题是所有 NP 类问题在多项式时间内可约化的 NP 问题, 若存在 NPC 问题有多项式算法, 则 P=NP。

NP-Hard NP-Hard 问题是所有 NP 类问题在多项式时间内可约化的问题(包括 NP 与非 NP 问题)。

[1]Sorting algorithm - Wikipedia https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

18.4.2 常见 NPC 问题

- $k-SAT, k>2$
- 哈密尔顿回路
- 最大团
- 最小点覆盖
- 旅行商问题
- 子集和问题
- 子图同构问题
- 整数线性规划问题
- 集合划分问题
- 最长简单回路问题

Chapter 19

杂项

19.1 思路与技巧 533
19.1.1 二分/三分 533
19.1.2 补集转化 533
19.1.3 莫队 533
19.1.4 分块 540
19.1.5 MITM 542
19.1.6 倍增 543
19.1.7 随机化 543
19.1.8 按位拆分 543
19.1.9 扫描线 543
19.1.10 差分 543
19.1.11 双指针法 543
19.1.12 优先队列维护长序列 544
19.1.13 集合选数最值问题 544
19.1.14 二进制分组 546
19.1.15 整体二分 547
19.1.16 cdq 分治 547
19.1.17 Kernelization 548
19.1.18 启发式合并 548
19.1.19 启发式分治 548
19.1.20 分段打表 548
19.1.21 注意事项/常见转化/思想 548
19.1.22 比赛注意事项 552
19.1.23 本节注记 554
19.2 卡常 554
19.2.1 取模 554
19.2.2 矩阵乘法 554
19.2.3 基于硬件的优化 557
19.2.4 位运算 558
19.2.5 搜索优化 560
19.2.6 数组清零 560
19.2.7 读入优化 560
19.2.8 快速乘法取模 563

19.1 思路与技巧

19.1.1 二分/三分

若目标函数具有单调性, 或者题目要求 ‘‘最大值最小/最小值最大’’, 一般考虑二分答案然后检查其合法性。

若目标函数为凸(尤其是一些计算几何题), 一般使用三分法判断函数的 ‘‘轮廓’’, 以达到缩小搜索范围的目的。使用斐波那契法(或者 0.618 法)在区间 $[0,1]$ 选取点 $\frac{f_{n-2}}{f_n}$, $\frac{f_{n-1}}{f_n}$ 作为比较点或许会更玄学。对于整数三分, 在范围缩小到 6 以内时暴力枚举。

19.1.2 补集转化

根据 ‘‘正难则反’’ 的哲学, 若正向思考不好解决, 则考虑它的反面。尤其在计数问题中考虑所求计数集合的补集。此外在并查集中引入补集的概念也是个不错的思考方向。

19.1.3 莫队

对于离线的 (修改 $+$) 查询问题, 可以使用莫队算法来维护一些难以合并维护的信息(比如区间颜色数)。

其思想是在能够快速对当前区间的左右端点进行移动并维护相应信息的情况下, 尽可能减少端点的移动距离。最优移动路径的计算可表示为二维(三维)曼哈顿距离最短哈密尔顿路径问题。一般使用简单的分块思想来达到较好的复杂度(~~O(能过)~~)。

19.1.3.1 例子

若能 $O(1)$ 对端点进行移动, 且区间大小与查询/修改量同阶(设为 n), 则对查询所涉及到的所有区间, 对每个位置每 $\sqrt{n}$ 分块编号, 然后对查询以左端点所在块编号为第一关键字, 右端点位置为第二关键字排序(若需要支持修改操作, 则第二关键字为右端点块所在编号, 第三关键字为时间)。处理询问时先移动到指定区间(若需要修改则移动到指定时间), 然后统计该询问的答案。

对于仅询问的问题, 该方法的复杂度为 $O(n\sqrt{n})$。当左端点在 $\sqrt{n}$ 个块时, 每块右端点最多移动 n; 而当左端点跨块转移时, 右端点最多移动 n, 所以右端点移动 $O(n\sqrt{n})$。由于查询量与区间大小同阶, 且左端点单次最多移动 $2\sqrt{n}$, 左端点的移动也是 $O(n\sqrt{n})$ 的。

19.1.3.2 分块大小

一般思路是推出各部分的复杂度, 然后使用均值不等式调到最优复杂度。带修改时注意查询移动与修改移动复杂度的平衡。

区间大小与查询量不同阶 设区间大小为 n, 询问规模为 m, 块大小为 k。右端点块内转移与跨块转移数为 $O(\frac{n}{k}\cdot n)$, 左端点转移数为 $O(mk)$, 利用均值不等式可得当 $k=\frac{n}{\sqrt{m}}$ 时取得最优复杂度 $O(n\sqrt{m})$。

修改 $+$ 查询

- 左端点移动次数 $O(nk)$;
- 右端点移动次数 $O(nk)$;
- 修改移动次数 $O(\left(\frac{n}{k}\right)^2\cdot n)$;

总复杂度为 $O(nk+\left(\frac{n}{k}\right)^2\cdot n)$, 令其导数为 0, 可得 $k=n^{\frac{2}{3}}$ 时最优, 时间复杂度为 $O(n^{\frac{5}{3}})$。

19.1.3.3 树上莫队

该方法用于维护树的链上信息, 维护子树信息可以使用 Dsu On Tree, 参见第 11.4节。

一种思路将树映射为括号序 (不分 +-), 然后按照处理普通序列的方法来做。考虑点 x, y 所映射的区间(设 $L[x] < L[y]$), 若 x 是 y 的祖先(使用 LCA 判定), 则为 $[L[x]\ L[y]]$, 否则为 $[R[x], L[y]]$。为了抵消掉途经子树的贡献, 维护一个 vis 数组, 根据 vis 判断当前途经点是否在贡献中, 然后相应地加上或减去贡献, 取反 vis 值。**注意查询答案时对于 LCA 处要特判, 若 x 不是 y 的祖先, 则要临时加入 LCA, 查询完毕后删去。因为此时 LCA 的左右括号均不在区间内, 而从 x 到 LCA 的链全因右括号产生贡献, 从 LCA 到 y 的链全因左括号产生贡献。**

另一种思路是对树进行分块。分块算法参考第 19.1.4节。分块后按照左右端点所在块的编号排序, 然后查询时在树上跑。

例题:WC2013 糖果公园 [1]

代码如下:

```cpp
// P4074
#include <algorithm>
#include <cmath>
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 100005;
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int d[size], p[size][17], L[size], R[size],
    id[size * 2], icnt = 0;
void DFS(int u) {
    for(int i = 1; i < 17; ++i)
        p[u][i] = p[p[u][i - 1]][i - 1];
    L[u] = ++icnt;
    id[icnt] = u;
```

[1] https://www.luogu.org/problemnew/show/P4074

```
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p[u][0]) {
            d[v] = d[u] + 1;
            p[v][0] = u;
            DFS(v);
        }
    }
    R[u] = ++icnt;
    id[icnt] = u;
}
int getLCA(int u, int v) {
    if(d[u] < d[v])
        std::swap(u, v);
    int delta = d[u] - d[v];
    for(int i = 0; i < 17; ++i)
        if(delta & (1 << i))
            u = p[u][i];
    if(u == v)
        return u;
    for(int i = 16; i >= 0; --i)
        if(p[u][i] != p[v][i])
            u = p[u][i], v = p[v][i];
    return p[u][0];
}
int bid[size * 2], v[size], w[size], c[size],
    ccnt[size], tmpc[size];
bool flag[size];
Int64 ans = 0;
void modify(int u) {
    flag[u] ^= 1;
    int t = c[u];
    if(flag[u])
        ans += v[t] * asInt64(w[++ccnt[t]]);
    else
        ans -= v[t] * asInt64(w[ccnt[t]--]);
}
struct Modify {
    int u, col, pcl, t;
} M[size];
void modify(int u, int dst) {
    bool t = flag[u];
    if(t)
        modify(u);
    c[u] = dst;
    if(t)
        modify(u);
}
struct Query {
    int l, r, id, lca, t;
```

```
    bool operator<(const Query& rhs) const {
        if(bid[l] != bid[rhs.l])
            return l < rhs.l;
        if(bid[r] != bid[rhs.r])
            return r < rhs.r;
        return t < rhs.t;
    }
} Q[size];
Int64 res[size];
int main() {
    int n = read();
    int m = read();
    int q = read();
    for(int i = 1; i <= m; ++i)
        v[i] = read();
    for(int i = 1; i <= n; ++i)
        w[i] = read();
    for(int i = 1; i < n; ++i) {
        int u = read();
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    DFS(1);
    int siz = pow(icnt, 2.0 / 3.0);
    for(int i = 1, cp = 1; cp <= icnt; ++i) {
        for(int j = 0; j < siz && cp <= icnt;
            ++j, ++cp)
            bid[cp] = i;
    }
    for(int i = 1; i <= n; ++i)
        tmpc[i] = c[i] = read();
    int mcnt = 0, qcnt = 0;
    for(int i = 1; i <= q; ++i) {
        if(read()) {
            int u = read();
            int v = read();
            if(L[u] > L[v])
                std::swap(u, v);
            int lca = getLCA(u, v);
            Query& cur = Q[qcnt++];
            cur.l = (lca == u ? L[u] : R[u]);
            cur.r = L[v];
            cur.t = i;
            cur.id = qcnt;
            cur.lca = (lca == u ? 0 : lca);
        } else {
            Modify& cur = M[++mcnt];
            cur.u = read();
            cur.col = read();
```

```
            cur.t = i;
            cur.pcl = tmpc[cur.u];
            tmpc[cur.u] = cur.col;
        }
    }
    std::sort(Q, Q + qcnt);
    M[0].t = 0;
    M[mcnt + 1].t = 1 << 30;
    int ct = 0, cl = 1, cr = 0;
    for(int i = 0; i < qcnt; ++i) {
        while(M[ct + 1].t < Q[i].t) {
            ++ct;
            modify(M[ct].u, M[ct].col);
        }
        while(M[ct].t > Q[i].t) {
            modify(M[ct].u, M[ct].pcl);
            --ct;
        }
        while(cr < Q[i].r)
            modify(id[++cr]);
        while(cr > Q[i].r)
            modify(id[cr--]);
        while(cl < Q[i].l)
            modify(id[cl++]);
        while(cl > Q[i].l)
            modify(id[--cl]);
        if(Q[i].lca)
            modify(Q[i].lca);
        res[Q[i].id] = ans;
        if(Q[i].lca)
            modify(Q[i].lca);
    }
    for(int i = 1; i <= qcnt; ++i)
        printf("%lld\n", res[i]);
    return 0;
}
```

19.1.3.4 奇偶排序优化

常规的查询区间排序是这样的:

```
0   bool operator<(const Query& rhs) const {
        return bid[l]!=bid[rhs.l]?l<rhs.l:r<rhs.r;
    }
```

这里有一个更好的排序方法:

```
0   bool operator<(const Query& rhs) const {
        return bid[l]!=bid[rhs.l]?l<rhs.l:
            (bid[l]&1?r>rhs.r:r<rhs.r);
    }
```

也就是使两个相邻块内的询问右端点移动方向不同。将右端点移动路径由 “[左-> 右]->[左-> 右]” 改为 “[左-> 右]->[右-> 左]”，减少了跨块转移的步数。注意块编号从偶数开始。

该方法参考了洛谷日报第 48 期 codesonic 的文章[2]与 RabbitHu 的博客[3]。

19.1.3.5 回滚莫队

回滚莫队适用于莫队的删除操作无法快速地更新答案的情况（比如维护最值，使用可删堆无法保证更新复杂度）。既然删除操作不好处理，干脆仅使用加入操作维护当前区间。考虑朴素区间排序方式，在左端点块标号相同的情况下，右端点单调递增，而左端点所在的块大小为 $O(\sqrt{n})$。那么可以将左端点所在块单独处理，块右边的端点移动仅有插入操作，总时间复杂度 $O(n\sqrt{n})$。每次处理完右边后，暴力枚举查询区间落在左端点所在块内的元素尝试更新答案（更新次数最多为 $O(\sqrt{n})$，对于维护最值问题可以将当前答案单独存储而不是全局修改，这样就不会影响到下一次查询），然后消除这些元素的影响（回滚莫队使其不影响最值的维护，其它数据很容易维护）。对于左右端点在同一块内的特殊情况，由于块大小为 $O(\sqrt{n})$，直接暴力维护。

例题「JOISC 2014 Day1」历史研究 回滚莫队经典题，参考代码：

```
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
typedef long long Int64;
#define asInt64 static_cast<Int64>
const int size = 100005;
int A[size], B[size], C[size];
Int64 cans;
void add(int pos, Int64& ans) {
    int col = A[pos];
    Int64 val = asInt64(++C[col]) * B[col];
    if(val > ans)
        ans = val;
}
Int64 query(int l, int r) {
```

[2]莫队算法初探 - #include<codesonic> https://www.luogu.org/blog/codesonic/Mosalgorithm

[3]胡小兔的良心莫队教程：莫队、带修改莫队、树上莫队 - 胡小兔 https://www.cnblogs.com/RabbitHu/p/MoDuiTutorial.html

```
    Int64 ans = cans;
    for(int i = l; i <= r; ++i)
        add(i, ans);
    for(int i = l; i <= r; ++i)
        --C[A[i]];
    return ans;
}
struct Queue {
    int l, r, bid, id;
    bool operator<(const Queue& rhs) const {
        return bid == rhs.bid ? r < rhs.r :
                                bid < rhs.bid;
    }
} Q[size];
Int64 ans[size];
int main() {
    int n = read();
    int q = read();
    for(int i = 1; i <= n; ++i)
        A[i] = read();
    memcpy(B + 1, A + 1, sizeof(int) * n);
    std::sort(B + 1, B + n + 1);
    int siz = std::unique(B + 1, B + n + 1) - (B + 1);
    for(int i = 1; i <= n; ++i)
        A[i] = std::lower_bound(B + 1, B + siz + 1,
                                A[i]) -
            B;
    int bsiz = sqrt(n) + 1;
    for(int i = 1; i <= q; ++i) {
        Q[i].l = read();
        Q[i].r = read();
        Q[i].id = i;
        Q[i].bid = Q[i].l / bsiz;
    }
    std::sort(Q + 1, Q + q + 1);
    Q[0].bid = -1;
    int bend, cr;
    for(int i = 1; i <= q; ++i) {
        if(Q[i].bid != Q[i - 1].bid) {
            memset(C + 1, 0, sizeof(int) * siz);
            cans = 0;
            cr = (Q[i].bid + 1) * bsiz;
            bend = cr - 1;
        }
        while(cr <= Q[i].r)
            add(cr++, cans);
        ans[Q[i].id] =
            query(Q[i].l, std::min(bend, Q[i].r));
    }
    for(int i = 1; i <= q; ++i)
```

```
        printf("%lld\n", ans[i]);
    return 0;
}
```

对于删除容易插入难的操作也可以这样做。

上述内容参考了 yashem66 的博客[4]。

19.1.4 分块

19.1.4.1 序列分块

一般将序列分为多块, 维护块内信息, 区间查询/修改时整块处理, 左右剩余元素暴力。块的大小根据具体情况而定(整块/零散复杂度平衡, 修改/查询复杂度平衡)。**一定要考虑区间端点在同一块内的情况。**

实现时块内暴力共有 3 种情况, 最好抽象出一个函数专门解决此类情况。

血泪史:THUWC2019 D1T1 中我使用了分块做法,但是先前没发现块在边界的情况要特判,即 1 与 n 虽然不是理论块的端点但是是实际块的端点,导致两端的块被重复统计,浪费了不少调试时间。因此在左右端在边界上时需强制不统计左右端。

带插入序列分块 例题:数列分块入门 6 by hzwer

插入操作集中在同一块内容易被卡, 因此可以选择每插入 $\sqrt{n}$ 个数将所有数重新分块, 每次重构的复杂度为 $O(n)$, 保证了块的大小相对平均, 并且重构总复杂度与查询复杂度相同。

不知为何每查询 + 修改 $\sqrt{n}$ 次重建比插入 $\sqrt{n}$ 次快得多。

区间众数问题 例题:数列分块入门 9 by hzwer

定理 19.1 有两个可重集合 A, B,记 $mode(A)$ 为 A 的众数,则 $mode(A \cup B) \in \{mode(A)\} \cup B$。

证明: 若 $mode(A \cup B) \neq mode(A)$ 且 $mode(A \cup B) \notin B$, 则 $mode(A \cup B) \in A$, $mode(A \cup B)$ 在 A 中的出现次数肯定不超过 $mode(A)$ 的出现次数, 与 $mode(A \cup B)$ 自身定义矛盾。

这个定理启示了我们可以维护整块(注意是所有整块连起来的块)的众数, 剩下的可能的众数一定在不完整块中出现, 枚举非完整块内的数字查询其出现个数。

考虑仅询问的情况, 首先预处理数组 $A[i][j]$ 表示第 i 块到第 j 块的众数, 枚举 i 向后扫记录扫描区间内每种数(需要提前离散化)的个数可以在 $O(n\sqrt{n})$ 内处理完毕。对于非完整块, 由于它们的大小不超过 $2\sqrt{n}$, 根据定理可以枚举每个数查询其在查询区间内的出现次数($O(n \lg n)$ 预处理每种数出现的位置, $O(\lg n)$ 二分查找区间内的位置, 这种方式以根号大小分块仍不为最优, 最优分块大小为 $\sqrt{\frac{n}{\lg n}}$。当然也可以使用分块预处理 $B[i][j][k]$ 表示第 i 块的第 j 个位置前等于 k 的位置数, 做到 $O(n\sqrt{n})$ 预处理, $O(1)$ 查询)。

修改操作留坑待补。

上述内容参考了陈立杰的《区间众数解题报告》。

[4]BZOJ4241 历史研究(分块回滚莫队-教程向)https://blog.csdn.net/qq_33330876/article/details/73522230

19.1.4.2 树分块

王室联邦分块法 DFS 遍历儿子时子树节点数累积大于等于 B, 则将其当做一块。其余节点并入父亲所在块, 保证块的大小 $\in [B, 3B]$, 且块的直径不超过 B, 但块不连通。

代码如下:

```
// P2325
#include <cstdio>
int read() {
    int res = 0, c;
    do
        c = getchar();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = getchar();
    }
    return res;
}
const int size = 1005;
struct Edge {
    int to, nxt;
} E[size * 2];
int last[size], cnt = 0;
void addEdge(int u, int v) {
    ++cnt;
    E[cnt].to = v, E[cnt].nxt = last[u];
    last[u] = cnt;
}
int B, st[size], top = 0, ccnt = 0, rt[size],
                  col[size];
void DFS(int u, int p) {
    int pos = top;
    for(int i = last[u]; i; i = E[i].nxt) {
        int v = E[i].to;
        if(v != p) {
            DFS(v, u);
            if(top - pos >= B) {
                rt[++ccnt] = u;
                while(top != pos)
                    col[st[top--]] = ccnt;
            }
        }
    }
    st[++top] = u;
}
int main() {
    int n = read();
    B = read();
    for(int i = 1; i < n; ++i) {
        int u = read();
```

```
        int v = read();
        addEdge(u, v);
        addEdge(v, u);
    }
    DFS(1, 0);
    while(top)
        col[st[top--]] = ccnt;
    printf("%d\n", ccnt);
    for(int i = 1; i <= n; ++i)
        printf("%d ", col[i]);
    puts("");
    for(int i = 1; i <= ccnt; ++i)
        printf("%d ", rt[i]);
    puts("");
    return 0;
}
```

size 分块法 DFS 时若父亲所在块还未到达指定 size, 将自身加入; 否则新开一个块。保证块的大小, 连通性还有直径大小。

树分块相关内容参考了 nimphy 的博客[5]。

19.1.4.3 平衡修改与查询的复杂度

例如对于规模为 n 的元素, 修改复杂度为 $O(n \lg n)$(比如暴力重建), 而查询复杂度为 $O(\lg n)$。此时可以考虑每 $\sqrt{n}$ 个元素分块, 单次修改/查询复杂度均为 $O(\sqrt{n} \lg n)$。

19.1.5 MITM

其主要思想是双向搜索(左 + 右/BSGS), 然后使用 hash 表来查询满足题意的集合。双向 BFS 搜索可以有效地减少状态数。

例题: Luogu P2324 [SCOI2005] 骑士精神

暴力搜索的状态数为 3^{15} 数量级, 但是双向 BFS 搜索的状态数为 $2 * 3^8$ 数量级, 可以通过该题。每轮迭代中, 两边各走一步, 同时进行检测, 这样做找到一组可行解就可以返回了。

19.1.5.1 经典模型

- 有向图模型: 给定有向图 G, 求点 A 到点 B 长度为 L 的路径数。

  解法: 从两个起点开始搜索长度为 $L/2$ 的路径, 使用散列表记录答案。

- **方程模型**: 求 $\sum_{i=1}^{n} f_i(x_i) = c$ 的方案数。

  解法: 将未知数分为两组枚举。

- 对于不指定始终点的题目, 考虑 "从中间出发"。

事实上并不一定要均匀分组, 不均匀分组有时能获得更优秀的复杂度。

该内容参考了国家集训队 2013 论文集乔明达的《搜索问题中的 meet in the middle 技巧》。

[5]【初识】树上分块 - nimphy https://www.cnblogs.com/hua-dong/p/8275227.html

19.1.6 倍增

通过多一个 lg 预处理跳跃 2^k 步的信息, 以达到 lg 级快速移动的目的。

19.1.6.1 倍增优化连边

有时需要从一个点到树上的一条链连边, 那么可以记 $P[i][j]$ 表示从点 i 开始往上 2^j 层的链对应的节点, 然后将其连无用边到子链。这样可以保证每次连边是 $O(\lg n)$ 的。

19.1.7 随机化

对于一些时间复杂度为期望复杂度的算法, 需要对数据顺序随机化来避免被卡。

对于枚举点对求最值的问题, 在允许的运行时间内随机生成点对并计算也可以骗到不少分。

19.1.8 按位拆分

对于位运算相关问题, 若位与位之间独立, 则可以考虑按位拆分计算, 这种做法可以降低时间复杂度, 简化代码。

19.1.9 扫描线

若某条件仅存在于一个区间/时间段中, 则可在起点与终点的后一点处打添加/删除标记, 查询时先处理完当前时间的所有添加/删除的标记, 再计算答案。对于一些计算几何题也可以考虑使用该方法。

19.1.9.1 逆扫描线

若需要维护每个时间段内元素集合的信息, 带有插入和删除操作, 可以考虑将其看做元素存在于一段时间区间, 使用线段树(分治)维护区间插入。

19.1.10 差分

- 对于离线区间加法, 可以在起点与终点后一点处打标记, 最后做一遍前缀和。
- 对于序列相邻位置的不等式约束, 有时也可以使用差分来转化问题。
- 对于具有单调性的序列, 比如前缀/后缀最值, 考虑差分后讨论修改位置。

19.1.11 双指针法

在区间最值问题中, 若区间移动存在单调性(即对于每个左端点所对应的最优右端点是单调的), 则对于一个固定的左端点, 使右端点移动到最优位置, 然后将左端点移动一格, 继续移动右端点。此法的复杂度为 $O(n)$。在两个区间上的组合问题也是如此(比如凸包矢量和问题)。

19.1.12 优先队列维护长序列

若要取出超长/无穷序列的前 k 小值, 且序列中较大的值的方案可以由较小的值的方案构造, 就可以预先加入原始的 (不能被其它方案构造) 方案到优先队列中, 取出时加入其后继方案到优先队列中。此法可保证优先队列的规模与算法时间复杂度在可接受的范围内。

有时甚至可以直接使用预排序 + 队列来代替优先队列 (即 $a \to a', b \to b', a < b \Rightarrow a' < b'$ 时)。

更加形式化的描述: 这些方案构造出了一个 DAG, 如果某个方案被选中, 它的前驱必定被选中。每次选取一个方案后将它的后继加入优先队列。如果这个 DAG 是一棵树, 就不需要去重, 最好构造出可树形转移的方案表示。

19.1.13 集合选数最值问题

19.1.13.1 类型一

给定 n 个可重集合, 在每个集合中选取一个数, 求这些选择方案中数的和(积)前 k 大(小)值。

当 $n = 2$ 时, 沿用上面的做法可以解决。但是当方案的维数增加时, 这种方法就不再有效了。思考的方向仍然时构造一棵树, 满足每个点只有一个前驱, 以及较少的常数个后继。

首先 DAG 的根肯定是每个集合都贪心选择最大值。对每个集合内的元素排序, 可以用 $(p_1, p_2, \cdots, p_n)$ 表示集合 i 选择第 p_i 个数, 但是这样的状态不好转移, 需要去重且后继数过多。考虑当前变动集合 i 的选择, 前面的集合全部固定, 后面的集合全部取最大值。那么可以使用状态 (i, j, s) 表示当前改变集合 i, 选择第 j 大的数, 该方案的价值为 s(这样就不必存储前面集合的选择方案)。然后得到如下后继:

- 当前集合后移: 若 $j < |S_i|$, 存在后继 $(i, j+1, s - A[i][j] + A[i][j+1])$;
- 改变下一个集合: 若 $i < n, 2 \le |S_{i+1}|$, 存在后继 $(i+1, 2, s - A[i+1][1] + A[i+1][2])$;
- 注意到变换下一个集合时当前集合选择的数肯定不是最大值, 但存在这种方案。所以指定当 $j = 2, i < n, 2 \le |S_{i+1}|$ 时当前集合选择最大值, 然后改变下一个集合。即存在后继 $(i+1, 2, s - A[i][2] + A[i][1] - A[i+1][1] + A[i+1][2])$。此时并不能保证其后继不大于前驱, 对集合以 $A[i][1] - A[i][2]$ 为关键字升序排序就可以解决这个问题。

这种状态设置方式保证了前驱大于等于后继且前驱唯一, 同时后继数可控而少。

注意如果以贪心选取最大值的方案为根, 第三种后继的假设不成立, 会出现重复方案。因此要以状态 $(1, 1, s_2)$ 为 DAG 根, 最大值单独统计。

例题: 第十三届北航程序设计竞赛预赛 M 最优卡组

```
0 #include <algorithm>
  #include <cstdio>
  #include <queue>
  #include <vector>
  int read() {
      int res = 0, c;
      do
          c = getchar();
      while(c < '0' || c > '9');
      while('0' <= c && c <= '9') {
10        res = res * 10 + c - '0';
          c = getchar();
      }
```

```
    return res;
}
typedef long long Int64;
const int size = 300005;
struct Set {
    std::vector<int> A;
    int key;
} S[size];
int id[size];
struct State {
    int i, j;
    Int64 sum;
    State(int i, int j, Int64 sum)
        : i(i), j(j), sum(sum) {}
    bool operator<(const State& rhs) const {
        return sum < rhs.sum;
    }
};
bool cmpG(int a, int b) {
    return a > b;
}
bool cmpS(int a, int b) {
    return S[a].key < S[b].key;
}
int main() {
    int n = read();
    int k = read();
    Int64 sum = 0;
    for(int i = 1; i <= n; ++i) {
        int c = read();
        std::vector<int>& A = S[i].A;
        A.resize(c);
        for(int j = 0; j < c; ++j)
            A[j] = read();
        std::sort(A.begin(), A.end(), cmpG);
        if(c == 1)
            S[i].key = 1 << 30;
        else
            S[i].key = A[0] - A[1];
        sum += A[0];
        id[i] = i;
    }
    std::sort(id + 1, id + n + 1, cmpS);
    std::priority_queue<State> heap;
    if(S[id[1]].A.size() == 1) {
        heap.push(State(1, 0, sum));
    } else {
        --k;
        printf("%lld ", sum);
        heap.push(State(1, 1, sum - S[id[1]].key));
```

```
    }
    for(int i = 1; i <= k; ++i) {
        State cur = heap.top();
        heap.pop();
        printf("%lld ", cur.sum);
        int cs = id[cur.i], ns = id[cur.i + 1];
        if(cur.j + 1 < S[cs].A.size())
            heap.push(State(cur.i, cur.j + 1, cur.sum -
                                S[cs].A[cur.j] +
                                S[cs].A[cur.j + 1]));
        if(cur.i < n && S[ns].A.size() >= 2) {
            heap.push(State(cur.i + 1, 1,
                            cur.sum - S[ns].key));
            if(cur.j == 1)
                heap.push(State(cur.i + 1, 1, cur.sum +
                                    S[cs].key -
                                    S[ns].key));
        }
    }
    return 0;
}
```

19.1.13.2 类型二

给定一个可重集合, 选择 m 个数作为一种方案, 求价值前 k 大的方案。

同样使用按位更改的方法, 不过要按顺序记录每个被选数的位置。这样的状态转移是一个 DAG, 存在重复状态。可以再加一维 i 表示 $i+1, \cdots m$ 的位置都被确定, 当前状态转移时要么将锁定位置前移并修改, 要么修改位置 i 选择的数。

特殊情况 若方案的价值定义为它们的积且存在负元素, 可以考虑将正负数分开计算, 然后枚举负数个数, 根据负数个数的奇偶性决定取负数积的绝对值的前 k 大/小值。最后使用类似双指针法根据计算出的正负数两个堆得到所需答案。

例题:SGU 421 k-th Product

该内容参考了 tkandi 的博客[6]。

19.1.14 二进制分组

对于在线向序列右端插入元素的操作 (不修改/删除), 可以按照最大规模固定规模分块后建块, 询问时对每块暴力查询。假设重建块的复杂度为 $O(n \lg n)$, 查询复杂度为 $O(\lg n)$, 询问量 q 与元素最大规模 n 同阶。离线问题则可以考虑对时间分治, 当然不管哪种方法都要求插入的贡献独立。

考虑将当前序列长度进行二进制拆分, 然后令每个拆分值对应一个块的大小。即新加入一个元素作为独立块后, 不断检查块序列最右端两个块大小是否相等, 相等则合并为一块并重建(注意要先标记范围再合并(可以 $O(1)$ 得到范围, 见下文), 每次插入最多合并一次)。此法保证了查询时块的数目是 $\lg n$ 的, 比固定大小分块算法更优。

接下来证明插入复杂度:

[6]集合选数最值一类问题 https://www.cnblogs.com/tkandi/p/9375509.html

插入第 k 个元素后, 需要合并的元素数为 $lowbit(k)$, 设 n 为 2 的幂, 有

$$\begin{aligned} T(n) &= \sum_{k=1}^{n} lowbit(k) \cdot \lg lowbit(k) \\ &= \sum_{k=1}^{\lg n} \frac{n}{2^{k+1}} \cdot k \cdot 2^k \\ &\leq \sum_{k=1}^{\lg n} n \lg n \\ &= O(n \lg^2 n) \end{aligned}$$

根据题目的需要可以选用 k 进制分组以优化时间复杂度。

19.1.15 整体二分

对于某些离线问题, 若其询问仅一种且可二分, 整体二分是个不错的选择。其思想是将所有询问的二分操作放在一起, 分治将询问分组到两个区间中。分治时可以顺便处理修改操作。

记整体二分主过程为 $solve(beg, end, l, r)$, 其中二分询问/查询为 $[beg, end)$, 二分答案范围为 $[l, r]$。

步骤如下:

1. 若区间元素为空则返回;
2. 若 $l == r$, 设置区间内元素的答案为 l 并退出;
3. 令 $m = (l + r)/2$;
4. 施加 $[l, m]$ 内的修改, 累积到每个询问的贡献(注意要保证这部分的复杂度与区间长度相关);
5. 对于修改按照位置分为左右集合, 对于询问按照当前累积贡献与目标值的关系将其分为 2 个集合;
6. 递归分治左右集合。

注意修改与查询之间也是有时间先后关系的, 可以调用 std::stable_partition 完成稳定划分。

19.1.16 cdq 分治

cdq 分治用来解决偏序问题(支持动态偏序问题, 但仍然要求离线)。

步骤如下 ($solve(l, r)$):

1. 若区间元素为空则返回;
2. 令 $m = (l + r)/2$;
3. 递归分治处理左右集合;
4. 计算一边集合对另一边集合的影响(比如左边元素对右边元素答案的影响, 左边修改对右边查询的影响, 有时左右元素会互相影响)。

对于静态偏序问题, 可以将第一维当做时间, 转化为动态偏序问题。对于动态偏序问题, 保证第一维按照时间排序, 递归合并时边计算影响边进行归并排序, 使其排序后位置从小到大, 修改优先于查询(要保证两集合内部元素的时间先后关系, 即排序要稳定)。

19.1.17 Kernelization

参见https://www.zhihu.com/question/272303098/answer/367368615。

19.1.18 启发式合并

如果某个区间内需要的数据结构可以用子区间的数据结构合并, 且规模与区间长度成正比, 就可以考虑使用启发式合并。每次合并时将较小的数据结构拆开, 插入大的数据结构中。每合并一次至少会使数据结构扩大一倍(针对较小数据结构而言), 合并次数为 $O(\lg n)$, 插入次数为 $O(n \lg n)$。此法适用于平衡树、链表等只支持单点插入的数据结构, 线段树则有另外一套简单的合并方法。

19.1.19 启发式分治

区间 $[l, r]$ 是否符合条件与区间的特殊值相关, 求符合条件的区间数。

如果区间中的某个数的特殊性可以决定跨越它的区间是否符合条件, 那么就可以从两边开始向中间寻找这个数, 然后递归处理被划分的两个区间。每次划分的复杂度取决于左右两端哪端更快找到特殊值, 时间复杂度 $O(n \lg n)$。实际时间复杂度还取决与跨数区间的处理复杂度, 一般使用可持久化数据结构预处理整个区间, 然后枚举较小区间的端点查询与较大区间的贡献, 一般时间复杂度 $O(n \lg^2 n)$。

例题:

- UVA1608/[Cerc2012]Non-boring sequences
- 「Luogu4755」Beautiful Pair
- 「LOJ6198」谢特 (SA 的 height 数组的性质使该方法很适用于此种问题)

上述内容参考了 DSL_HN_2002 的博客[7]。

19.1.20 分段打表

需求:少量大规模单点查询, 查询上界已知, 支持快速转移。比如求 $1e9$ 范围内的模意义阶乘。

可以写一个打表程序, 每隔 $1e5$ 输出一个点, 然后将其导入源程序, 查询时寻找最近的点, 从该点开始转移。

19.1.21 注意事项/常见转化/思想

- 求解区间相交/包含/相离问题时首先判断这三种情况是否合法。若相交不合法, 可以考虑将区间缩点。对于序列问题考虑其置换 $b_{a_i} = i$。(WYX'S BLOG [8])
- 「雅礼集训 2017 Day8」价:最小割建图时考虑与 inf 加减作为边权。
- 「TJOI / HEOI2016」字符串:前缀 $\rightarrow$ 后缀, 使用后缀系列算法解决。
- 删除 + 可离线 $\rightarrow$ 逆序加入
- 「LNOI2014」LCA、HNOI2015 开店:两点的 LCA 深度可以理解为一个点到根的链标记 +1, 统计另一个点到根的标记总和。

[7]「随笔」一种基于启发式思想的分治策略——启发式分裂 https://blog.csdn.net/DSL_HN_2002/article/details/81193576

[8][CTSC2018] 青蕈领主http://blog-wayne.com/2018/05/16/523/

- 「LibreOJ Round #6」花火：将点的编号与点值视为二维平面上的一个点 $P_i = (i, h_i)$，若交换 i, j 则减少了 P_i, P_j 矩形内的点数（不含边界）*2+1。可以推出 h_i 一定是前缀最大值，h_j 一定是后缀最小值，否则存在更优的矩形。求出前缀最大点集 L 与后缀最小点集 R 后，设选择的矩形为 (i, j)，考虑点 k 被包含在 (i, j) 内的条件。记 l 为前缀最大点集中在 k 左上方的点的最左位置，r 为后缀最小点集中在 k 右下方的最右位置，这两个均可以二分处理。那么条件为 $i \in [l, k-1], j \in [k+1, j]$，将问题转化为每个点 k 覆盖一个矩形，求点的最大覆盖值。使用扫描线 + 线段树解决。最后答案为逆序对-最大覆盖数 *2（可非相邻交换 1 次与减少 1 对逆序对抵消）。
- 「LibreOJ Round #11」Misaka Network 与任务：需要大量同指数快速幂且已知底数范围时使用线性筛预处理。
- 解题时先考虑不带修改的情况。
- 利用时间存储数据。
- THUWC2019 D2T1：对于统计树上所有路径的权值和问题，除了点分治外，一个比较简单的思路是考虑 DFS 统计一条边两边节点与这条边的贡献。由于在换边时点集变化比较大，可以考虑统计当前遍历的点/边与已遍历边/点的贡献。注意 DFS 从边进入子树与从边走出子树时边的权要修改（子树反向）。
- 最短路、网络流问题在时空复杂度无法承受时考虑动态加边。
- 询问参数局部修改时考虑从上一个最优解开始迭代。
- 区间范围没有保证 $l \le r$ 时要注意 swap。
- 节点到根的路径上的点权值 +1，统计某个点的权值。~~树链剖分 + 区间修改单点查询~~。使用树上差分直接在该点修改，查询时查询子树权值和，可以 $O(n)$ 预处理 $O(1)$ 查询，比原方法修改少 $\lg^2 n$，查询少 $\lg n$。链上操作类似。

  一般化 + 在线：链上节点权值 +1，统计某个点的权值。同样采用树上差分，在两端点处 +1，在 lca 和 lca 父亲处-1。单点修改与查询子树权值和使用树状数组解决。
- 多个卷积后的函数之和可以在点值乘法时就相加，最后只做一次 IDFT。
- 区间 gcd 问题：注意到 $gcd(a, b, c) = gcd(a, b-a, c-b)$，对原数组差分后仅需支持单点修改，单点查询和区间 gcd 操作。差分后可以支持区间加减操作。
- 支持有后效性的可加减区间操作，询问历史版本和：维护两个数组 A, B，在 A 上记录所有区间操作，在 B 上记录若该区间操作从 0 时刻开始与从现在开始的贡献差。若当前询问时刻为 t，则答案为 $t * A - B$。
- 无标号的方案数可转化为有标号的方案数解决，最后将答案除以每种元素个数的阶乘之积。
- 写题时忽视背景描述，读清题目，看一遍小样例，确定明白题意后想题。想题时先多推性质，然后每部分选择最简数据结构/算法实现，确定算法正确且复杂度正确后再敲代码。
- 对字符进行重映射时注意字典序。~~FWC2019D1T2 AGCT 把我最后的 40 分也给送子~~。
- 给一些元素 A 分配不重复元素 B：考虑枚举足够的元素 B，然后给两种元素连边跑二分图匹配/费用流。

- 要确保暴力算法/小数据特判的正确性。~~FWC2019D2T1 特判写炸 + 捆绑测试 =AC ->20 分~~
- 构造问题中要大胆猜想, 利用随机性质缩小搜索范围, 对于序列构造问题考虑排序贪心, 对于树的构造问题考虑菊花树/二叉树。
- DP: 考虑元素的组合是否要求有序。比较数据范围, 合并本质相同的元素, 降低时间复杂度。
- n 很大, 求第 n 项: 若答案是次数与其他项有关的关于 n 的多项式, 求出小数据答案后多项式插值。若答案是次数与 n 有关的多项式, 考虑矩阵快速幂。
- 每个物品有存在区间, 查询在某个时刻 t 的答案。一般使用离线 + 扫描线解决。若存在区间定长为 p, 则可以将原问题转化为查询 $(t-p,t]$ 出现的物品的贡献。记区间长度为 L, 在 L 上每 p 个时间取一个关键点, 可以发现任意查询区间都恰好覆盖一个关键点。预处理从每个关键点开始向左右 p 个时间的答案。查询时定位到对应关键点, 进而定位到预处理左右区间的答案, 快速合并得到答案。预处理的长度是 $O(L)$ 的。猫树也是这个思想。
- 如果离线处理时空间不够, 那就分多次离线。注意有些问题无法分割 (比如组合问题)。
- 对于区间内的数的最大值为定值的限制, 可以将其表示为区间内的数不大于该定值的方案扣除区间内的数不大于该定值-1 的方案。
- 矩阵快速幂时若时间复杂度不优则观察矩阵的性质。
- 与排序有关的问题: 考虑枚举某个数 x, 将整个序列表示为 01 序列。序列在全局下有序, 说明相邻的数之间都是有序的。
- 「HAOI2018」字串覆盖: 根据数据范围提示讨论子串大小, 不重叠子串长则出现次数少, 分两种情况使用不同方法处理。「雅礼集训 2017 Day1」字符串: $k*q$ 小于等于定值 $1e5$, 并且当 k 较小与 q 较小时都有较优的做法, 分情况处理。一般来说, 若存在两个量的乘积不超过定值, 并且存在两种高效算法分别主要与这两个量有关, 那么就可以考虑对询问分类处理。
- 最长公共子序列: dp 数组单调且相邻项之差不超过 1 时考虑对数组差分, 得到 01 序列, 然后上 bitset 加速转移。
- 对于对相邻项/行列/对称位置有约束的矩阵构造问题, 考虑两个方向的斜线以及分块递归构造。
- 注意 FFT 与 NTT 计算的是循环卷积, 可以利用此性质优化常数。
- 「LibreOJ Round #7」匹配字符串: 对于类似于 $f_i = af_{i-1} + bf_{i-m}$ 这类的表达式, 可以将其视作一个 DAG, 其中 i 向 $i+1$ 连一条权值为 a 的边, 向 $i+m$ 连一条权值为 b 的边, 定义一条路径的权值为边权之积。若 $f_0 = 1$, 则求 f_n 即为求从点 0 到点 n 的路径权值和。那么考虑枚举走 $i \to i+m$ 的边数, 使用可空隔板法, 有

$$f_n = \sum_{i=0}^{\lfloor \frac{n}{m} \rfloor} \binom{n-im+i}{i} a^{n-im} b^i$$

- 在模费马素数意义下, 由于其 φ 为 2 的幂, 且以任意整数为生成元做模意义乘法构成的群的阶都是它的因子。对于某个常底数, 它的阶可能极小, 可以根据指数分类计算。例如 2 模 65537 的循环节大小为 32。

- 做概率期望类 dp 时，若每一步都贪心选取最优方案，需要从后往前 dp。
- CF553E Kyoya and Train：dp 不好按照点转移，但发现时间 t 是递增的，状态可以拆成分层图，因此使用 cdq 分治 FFT 转移。
- $B'(x)A(x) = A'(x) \Rightarrow B(x) = \ln A(x)$。
- 求某个元素必选的最优方案：分别对前缀与后缀 dp，再组合 dp。
- 用无限次修改最优化某个量：寻找修改中的不变量。
- 统计区间或链上颜色数：考虑每个颜色的贡献，对每种颜色分别统计。
- 考虑是否仅依赖相邻位置的关系，序列 $\Rightarrow$ 相邻位，网格图 $\Rightarrow$ 相邻行。
- APIO2018 铁人两项：树上指定点集两两树上路径点权和：考虑每个点权的贡献。
- 当元素之间无先后顺序要求时，可以考虑按照元素的某个属性排序。
- 对于等比数列求和问题，若不存在乘法逆元，则考虑分治计算或使用矩阵记录历史版本和，再做快速幂。
- 树上某点的儿子的子树大小只有 $O(\sqrt{n})$ 种，可以合并以优化复杂度。
- 边在满足指定条件时连通：
  - 单向限制 $(w \le a)+$ 强制在线：重构树
  - 双向限制 $(a \le w \le b)+$ 可离线：扫描线 +LCT
- 连通块计数
  - 树上连通块个数 = 点数-边数（统计无向图连通块个数需要去环，根据题目需要以权值/编号为关键字删掉环上的某条边，一般使用 LCT 维护）
  - 网格图四连通块个数 = 格子数-1*2 矩形数-2*1 矩形数 +2*2 矩形数 + 环数
- 注意最小权最少连通块最多只有 n-1 条边（MST），在合并时可以使用 Kruskal 保留有用边。
- $\ln \frac{1}{1-x} = \sum_{i=1}^{\infty} \frac{x^i}{i}$，在做多项式 ln 前考虑生成函数的封闭形式是否可以直接计算。
- 最大化 k 个连续子段和（【POJ Challenge】生日礼物）：
  - 费用流：$S \to [1,n]$ 连边 $(1,0)$，$i \to i+1$ 连边 $(1, A[i])$，$[2,n+1] \to T$ 连边 $(1,0)$，控制最大流为 k。
  - 线段树模拟费用流：注意到此题费用流建图的结构比较简单，考虑增广的过程。选取最长增广路径相当于查询区间最大子段和，增广的过程相当于区间取反（下次如果选取了这一段就相当于退流），用线段树支持这两种操作。
    选取不超过 k 个正权区间即为答案（朴素费用流可以走空区间）。
  - 贪心 + 优先队列：首先将同号的缩点，然后去除两边的负数。考虑选取所有的正块，然后将块数降为 k。那么可以先将所有正块加入答案，再将所有块的绝对值加入优先队列，每次选取权值最小的删除，答案减去它的权值。删除正块相当于舍弃它，删除负块相当于选取它，然后可以合并两边的正块。注意一个块一旦被选择，它的两边都不能被选择。用一个链表维护：如果当前块不为边界，则保留左右块，将左右块节点删除，可能选择左右块节点优，需要支持后悔操作（模拟费用流一般都需要），权值为左右块权值和减去自身权值，将其加入优先队列中；否则左右块与自己都不能再被选择，将其从链表中删去。

- 对于整体取反操作, 可以同时维护两个方面的信息, 取反时直接 swap。
- 对于区间内选取最佳位置问题, 可以将询问按照左端点排序, 然后从右到左加入每个位置, 使用单调栈维护位置作为答案的区间, 二分查询回答询问。
- 对于复制数据结构的问题, 考虑复制的每个元素中待统计信息的占比, 修改时大部分的转移是相同的。
- 方差计算的转化:
$$\sum (x_i - \bar{x})^2 = \sum x_i^2 - n\bar{x}^2$$
- 考虑将一些依赖/包含关系描述为图/树/DAG 的形式。
- 当计数出现重复时, 考虑设出容斥系数函数 $f(x)$, 然后利用等式解出。

19.1.22 比赛注意事项

19.1.22.1 Linux/GCC 工具

鉴于 THUWC2019Day3 工程题的 CRC32 校验可以直接使用系统内置工具。。。

- hashsum/md5sum/shasum/cksum/crc32/sum:计算文件校验和
  注意校验文本时一定要使用不换行输出,即 echo -n xxx | xxxsum。
- python:本地高精度计算/binascii.crc32/hashlib.md5
- size:查看可执行文件静态分配内存大小
- diff:文件比较, 要加入-s-b 选项以忽略多余空格和换行, 完全相同时输出信息
- perf(Linux 不一定自带):底层性能分析
  主要性能指标为 IPC, Cache 命中率, 分支命中率。所以命令为 perf stat -e cache-misses,branch-misses,instructions,branches,cache-references, cpu-cycles ./a.out
- time:测量程序运行时间
- ulimit -s size:开栈

19.1.22.2 用于 Debug 的编译命令

- -ftrapv 有符号算术运算溢出检测(需要时再开, 因为有时候会利用自然溢出)
- -Wall 最高级别警告
- -Wextra 扩展警告
- -fsanitize=undefined 未定义行为检测
- -fsanitize=address 访问越界检测
- -ggdb3 生成适宜 GDB 的调试信息

19.1.22.3 进程信息查询

在/proc/self/status 文件中有程序运行信息, 其中 VmPeak 指示峰值虚拟内存。使用时在程序退出前将这个文件的内容重定向至 stderr。

19.1.22.4 代码注意事项

- 提前退出计算时注意数据清空:
  - setjmp/longjmp: 不要使用任何 RTTI 数据结构, 全部的 longjmp 由一个函数 earlyExit 调用, 函数内执行清空操作
  - break/return: 使用 RTTI 技术
  - throw/try-catch: 别用这个
- 个人喜欢使用但是 C++98 不支持的:
  - std::vector<>::data()
  - fmax/fmin
  - fma
  - <random>
  - <chrono>
  - 委托构造函数
- 不要使用 while(n−−) 这种语句, 宁可用 for(int i=0;i<n;++i) 代替。因为并不知道将来是否要用到 i 或 n, 还可能导致莫名其妙的错误。
- int 内二分时, 要注意 $l + r$ 是否溢出, 一般使用 $l + ((r - l) >> 1)$ 代替。
- 尽量不要使用与标准库函数相同的名称, 哪怕没有 using namespace std; (尤其是 STL, 编译器的错误信息能让人抓狂)。
- 边读入边处理时不要跳出循环, 若要跳出循环则需继续读入以跳过冗余数据。
- std::priority_queue<> 的清空使用.swap() 完成
- 删除 std::multiset 内的单个元素时要使用迭代器
- std::set 的插入最好使用 hint 与区间。
- (分段)打表时注意源代码不能超过 100KB。
- 不要使用 std::bitset 的 set 与 test, 它们会执行越界检查。不过 operator[] 的性能尚未验证, 因为它们的返回值是 bool 的 Proxy Class——std::bitset::reference。
- 跨平台 scanf/printf: 最保险的方式是手写输入输出。另一种方式是使用 <cinttypes> 提供的 PRI/SCN* 宏。例如不管 long long 输入输出是%lld 还是%I64d, 都可以使用 SCNd64/ PRId64 表示。
- 需要稳定划分时不能使用维护左区间 end 位置与 swap 操作实现, 必须使用额外缓冲区, 调用 std::stable_partition。
- set 与 map 的插入与删除操作不影响之前取得的迭代器的有效性。
- 调试输出时重定向至 stderr, 即使忘记删掉也可以降低风险。
- %c 输入时不会跳不可见字符, 可以在其前面加空格或者使用%s
- 使用 strtol/strtod 时, 要么一次性读入整个文件, 要么读完整段后再调用, 直接在缓冲区上搞可能会导致 RE 与读入错误。
- 避免出现 Undefined behavior 导致优化后程序错误。例如有符号整数溢出: 要利用自然溢出特性时强转至 unsigned 做加法。

19.1.23 本节注记

2013 年许昊然的国家集训队论文答辩《浅谈数据结构题的几个非经典解法——<Claymore> 命题报告》中提到了不少奇妙的解法。

19.2 卡常

19.2.1 取模

- 对于多个两整数乘积之和的取模(比如模意义下矩阵乘法), 可以设置一个阈值, (绝对值)超过该阈值才取模, 最后再做一次取模。这种方法在保证加法不溢出的情况下大幅减少取模次数, 同时将值存储在寄存器内访问速度更快, 并且 if 分支的命中概率小, 分支预测效率高。

```
const Int64 end=std::numeric_limits<Int64>
    ::max()-asInt64(mod-1)*(mod-1);
...
    for(int i=0;i<n;++i)
        for(int j=0;j<n;++j) {
            Int64 sum=0;
            for(int k=0;k<n;++k) {
                sum+=asInt64(A[i][k])*B[k][j];
                if(sum>=end)
                    sum%=mod;
            }
            C[i][j]=sum%mod;
        }
```

- 若可以肯定最终答案在整型范围内且只有加减运算, 可以允许暂时的加法溢出;
- 若模意义加减操作多, 则保证在所有计算过程中的数 $\in [0, mod)$。

```
int add(int a,int b) {
    a+=b;
    return a<mod?a:a-mod;
}
int sub(int a,int b) {
    a-=b;
    return a>=0?a:a+mod;
}
```

- 若模意义乘法操作多, 则仅保证中间数 $\in (-mod, mod)$, 没有必要 $\cdots = clamp(\cdots)$。在最后输出时 $clamp$。

19.2.2 矩阵乘法

不同优化下的矩阵乘法性能差异巨大, 下面记录一些常用的优化。

- 见 19.2.1第一点;
- 考虑访问矩阵时 cache 的连续性, 发现按照 i, j, k 访问时 $B[k][j]$ 的访问位置跳跃较大, cache 性能较低; 但是如果按照 i, k, j 的顺序计算, 就可以使 $C[i][j]$ 与 $B[k][j]$ 的访问位置连续, 提高访问速度;

- 在第 3 层循环内进行循环展开。
- 遇到稀疏矩阵时使用 ikj 枚举顺序, 提前判断 $A[i][k]$ 是否为 0。用这个方法能够获得跑过规模为 400 的矩阵快速幂的信仰。

性能测试代码:

```
#include <chrono>
#include <climits>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <random>
const int N = 500, mod = 1000000007;
using Clock = std::chrono::high_resolution_clock;
class Timer {
private:
    const char* func;
    Clock::time_point beg;

public:
    Timer(const char* func)
        : func(func), beg(Clock::now()) {}
    ~Timer() {
        printf("%s %.3lf ms ", func,
               (Clock::now() - beg).count() * 1e-6);
    }
};
typedef long long Int64;
#define asInt64(x) static_cast<Int64>(x)
int A[N][N], B[N][N], C[N][N], D[N][N];
void mulStandard() {
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            for(int k = 0; k < N; ++k)
                C[i][j] =
                    (C[i][j] +
                     asInt64(A[i][k]) * B[k][j]) %
                    mod;
}
const Int64 end = std::numeric_limits<Int64>::max() -
    asInt64(mod - 1) * (mod - 1);
void mulOptimizedMod() {
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j) {
            Int64 sum = 0;
            for(int k = 0; k < N; ++k) {
                sum += asInt64(A[i][k]) * B[k][j];
                if(sum >= end)
                    sum %= mod;
            }
```

```
            C[i][j] = sum % mod;
        }
    }
    Int64 E[N][N];
    void mulOptimizedCache() {
        memset(E, 0, sizeof(E));
        for(int i = 0; i < N; ++i) {
            Int64 *eib = &E[i][0], *eie = &E[i][N];
            for(int k = 0; k < N; ++k) {
                Int64 aik = A[i][k];
                int* bkj = &B[k][0];
                for(Int64 *eij = eib; eij != eie;
                    ++eij, ++bkj) {
                    *eij += aik * (*bkj);
                    if(*eij >= end)
                        *eij %= mod;
                }
            }
        }
        for(int i = 0; i < N; ++i) {
            Int64 *eib = &E[i][0], *eie = &E[i][N];
            int* cij = &C[i][0];
            for(Int64 *eij = eib; eij != eie; ++eij, ++cij)
                *cij = *eij % mod;
        }
    }
    void mulOptimizedUnfold() {
        const int step = 16;
        memset(E, 0, sizeof(E));
        for(int i = 0; i < N; ++i)
            for(int k = 0; k < N; ++k) {
                Int64 aik = A[i][k];
    #define Unit(x)                                \
        {                                          \
            E[i][j + x] += aik * B[k][j + x]; \
            if(E[i][j + x] >= end)                 \
                E[i][j + x] %= mod;                \
        }

                int j;
                for(j = 0; j < N - step; j += step) {
                    Unit(0) Unit(1) Unit(2) Unit(3);
                    Unit(4) Unit(5) Unit(6) Unit(7);
                    Unit(8) Unit(9) Unit(10) Unit(11);
                    Unit(12) Unit(13) Unit(14) Unit(15);
                }
                while(j < N) {
                    Unit(0);
                    ++j;
                }
```

```
#undef Unit
        }
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            C[i][j] = E[i][j] % mod;
}
#define test(func)                                          \
    {                                                       \
        memset(C, 0, sizeof(C));                            \
        {                                                   \
            Timer guard(#func);                             \
            func();                                         \
        }                                                   \
        puts(memcmp(C, D, sizeof(C)) == 0 ? "AC" : \
                                            "WA"); \
    }

int main() {
    puts("Generating matrix...");
    std::random_device rd;
    std::uniform_int_distribution<int> gen(0, mod - 1);
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            A[i][j] = gen(rd);
    for(int i = 0; i < N; ++i)
        for(int j = 0; j < N; ++j)
            B[i][j] = gen(rd);
    mulStandard();
    memcpy(D, C, sizeof(C));
    puts("Done.");
    test(mulStandard);
    test(mulOptimizedMod);
    test(mulOptimizedCache);
    test(mulOptimizedUnfold);
    return 0;
}
```

2000*2000 矩阵乘法性能测试结果如下(i7-4790K):
mulStandard 105793.233 ms AC
mulOptimizedMod 27042.733 ms AC
mulOptimizedCache 12499.686 ms AC
mulOptimizedUnfold 11096.677 ms AC
每个算法由上一个算法修改而来, mulOptimizedUnfold 使用了所有优化, 比原始算法快了接近 10 倍, 可见对矩阵乘法进行优化还是很划算的。

19.2.3 基于硬件的优化

- 循环展开:指定一个步长, 满步长区间硬编码, 剩余部分暴力。
- 多路并行: 在循环展开的同时避免修改同一变量, 即保持循环间的写独立, 这样避免 CPU 流水线被打断。

- Cache 优化:尽可能保证循环时访问位置连续。
- 尽可能地使用临时变量,这样可以保持数据在寄存器中,最后再写回数组。不仅减少了寻址时间,还能简化代码编写。
- 结构体对齐:安排结构体元素时按照对齐大小从大到小定义
- Cache 优化:大数组的每一维大小都不要是 2 的幂,尤其是状压 dp 和 FFT。因为 Cache 会存储额外信息。
- 寻址优化:高维数组寻址时使用指针代替。
- Cache 优化:在性能敏感的地方,使用指针存储儿子(当然还要考虑 if 判断 nullptr 的开销,当然也可以使用)。
- 尽可能使用引用而不是指针:当派生类有多个基类,在继承体系中向上转型时,由于空引用是非法的,程序只要计算指针的偏移;由于允许存在空指针,根据规定,空指针向上转型后仍然是空指针,需要一次特判。
- 手写位运算代替乘除法:有时编译器为了符合标准需要做一些额外工作。
- 使用最高效的整型:比如用 int 代替 bool,或者使用 C++11 新增的 int_fastXX_t

19.2.4 位运算

19.2.4.1 符号判断

$$sign(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0 \end{cases}$$

```
0 int sign(int x) {
      return (x>0)-(x<0);
  }
```

此法同样适用于浮点数符号的判断。

19.2.4.2 判断异号

```
0 bool flag=((x^y)<0);
```

19.2.4.3 绝对值

```
0 int iabs(int x) {
      int mask=x>>31;
      return (x+mask)^mask;
  }
```

若 x 为非负则为 $(x+0)\oplus 0 = x$,若 x 为负则为 $(x-1)\oplus 0\text{xffffffff} = -x$(注意有符号右移时高位补符号位)。

19.2.4.4 去末尾 1

$v\& = v - 1$, 即使 v 末尾的 $100\cdots$ 部分与 $011\cdots$ 按位与。

19.2.4.5 取末尾 0 的数量

首先使用 $w = v\& - v$ 取得最低位 1, 然后无符号乘以 De Bruijn Sequences 常数 0x077CB531U, 最后前五位与每种 w 一一对应。LUT 可预处理。

```
int countTZ(int x) {
    unsigned int bit=x&-x;
    return LUT[(bit*0x077CB531U)>>27];
}
```

64 位整数下用的常数是 0x07EDD5E59A4E28C2ULL。

19.2.4.6 子集枚举

一般使用如下写法:

```
for(int i=S;i;i=(i-1)&S)
```

可以理解为每次都做一次忽略 S 非零位的减法, 从全集开始枚举自然能够枚举所有子集。

父集枚举 将子集枚举对应操作取反, 注意要指定全集 end。

```
for(int i=S;i<=end;i=(i+1)|S)
```

19.2.4.7 右起连续的 0/1 取反

0->1:x|(x-1), 1->0:x&(x+1)

19.2.4.8 取右边连续的 1

$(x \oplus (x+1)) >> 1$

19.2.4.9 取最高位 1

```
int getHighest(int x) {
    for(int i=1;i<=16;i<<=1)
        x|=x>>i;
    return x^(x>>1);
}
```

其原理为倍增长度使最高位右边全置 1, 最后清空除最高位外的位。

19.2.4.10 大小写转换

由于 ASCII 码中大小写字母的比特位之间的特殊关系, 异或一个空格 (0b100000) 可以切换大小写。

19.2.4.11 整数 log2

一种写法是取最高位 1+ 取末尾 0 的数量。不过这种写法比较麻烦。$O(n)$ 预处理 + 查表虽然好写但又太浪费。考虑最大位数为 b, 满足 $2^b > n$, 可以预处理 $c = \lceil \frac{b}{2} \rceil$ 的表, 查询时先比较判断其位数是否小于 2^c, 若小于则直接查表, 否则查 $>> c$ 的值, 再加回 c。$O(\sqrt{n}) - O(1)$ 的复杂度绝对不会成为性能瓶颈。

以上算法基本参考了 Sean Eron Anderson 的文章[9]。

19.2.5 搜索优化

- 维护全局最优值, 尽可能剪枝;
- 对于一些计算几何题, 通过随机旋转坐标系 + 不正确的贪心来提高寻找最优解的速度(也可以作为预处理指导剪枝)。
- 在求权值最优的点对时, 使用随机算法骗分;
- 对于多次修改 + 全局询问的问题, 其最优解不会移动太远, 考虑从上一个最优解开始移动搜索。例如 [ZJOI2015] 幻想乡战略游戏。

19.2.6 数组清零

- 整个数组的清零可以使用 memset, 因为它的实现一般有循环展开/SIMD 优化。
- 若仅修改整个数组的部分数据, 可以重新扫一遍修改时的数据, 撤销修改操作/ 直接将对应位置置 0(这会影响到算法时间复杂度, 尤其是对于 Dsu On Tree/cdq 分治);
- 对于树状数组, 在模拟树状数组修改算法置零时, 若当前值为 0, 则直接退出, 因为接下来的值肯定都为 0(肯定在清除其他链上数时被清零了)。

```
0     void clear(int x) {
          while(x<=siz && A[x]) {
              A[x]=0;
              x+=x&-x;
          }
      }
```

- 对于 bool 数组无需清零, 使用 int 记录其最后一次被标记为 true 的时间戳 *timeStamp*, 若时间戳与当前时间戳相等则为 true, 将当前时间戳 +1 可实现 $O(1)$ 清零。对于其它值的存储也可以如此清零(std::pair< 时间戳, 实际数据 >)。不过对其它值的存储的实际性能不如直接记录修改位置清零快(每次访问都要检查一次时间戳)。

19.2.7 读入优化

被别人的时间优势刺激到了才入了 fread/fwrite 的坑。。。

下面的测试使用同一组整数与字符串转换的算法:

IOTest.h

```
0 int read() {
      int res = 0;
```

[9]Bit Twiddling Hacks http://graphics.stanford.edu/~seander/bithacks.html

```
    char c;
    do
        c = IO::getc();
    while(c < '0' || c > '9');
    while('0' <= c && c <= '9') {
        res = res * 10 + c - '0';
        c = IO::getc();
    }
    return res;
}
void writeImpl(int x) {
    if(x >= 10)
        writeImpl(x / 10);
    IO::putc('0' + x % 10);
}
void write(int x) {
    writeImpl(x);
    IO::putc('\n');
}
int main() {
    IO::init();
    int n = read();
    write(n);
    while(n--)
        write(read());
    IO::uninit();
    return 0;
}
```

数据生成器如下:

IOGen.cpp

```
#include <cstdio>
int getRandom() {
    static int seed = 2351432;
    return seed = seed * 48271LL % 2147483647;
}
int main() {
    const int size = 1 << 27;
    freopen("data/IO1.in", "w", stdout);
    printf("%d\n", size);
    for(int i = 1; i <= size; ++i)
        printf("%d\n", getRandom());
    return 0;
}
```

常规 getchar/putchar:31058 ms

Test A

```
#include <cstdio>
namespace IO {
```

```
    void init() {}
    void uninit() {}
    char getc() {
        return getchar();
    }
    void putc(int ch) {
        putchar(ch);
    }
}
#include "IOTest.hpp"
```

使用 8MB 自带缓冲区, 显式 fread/fwrite, 关闭内置缓冲区:9724 ms

Test B

```
#include <cassert>
#include <cstdio>
namespace IO {
    void init() {
        assert(setvbuf(stdin, 0, _IONBF, 0) == 0);
        assert(setvbuf(stdout, 0, _IONBF, 0) == 0);
    }
    const int size = 1 << 23;
    char in[size];
    char getc() {
        static char *S = in, *T = in;
        if(S == T)
            S = in, T = in + fread(in, 1, size, stdin);
        return S == T ? EOF : *S++;
    }
    char out[size], *S = out, *T = out + size;
    void putc(char ch) {
        if(S == T) {
            fwrite(out, size, 1, stdout);
            S = out;
        }
        *S++ = ch;
    }
    void uninit() {
        fwrite(out, S - out, 1, stdout);
    }
}
#include "IOTest.hpp"
```

重置内置缓冲区为 8MB, 使用全缓冲模式:23461 ms

Test C

```
#include <cassert>
#include <cstdio>
namespace IO {
    void init() {
        const int size = 1 << 23;
```

```
        assert(setvbuf(stdin, 0, _IOFBF, size) == 0);
        assert(setvbuf(stdout, 0, _IOFBF, size) == 0);
    }
    void uninit() {}
    char getc() {
        return getchar();
    }
    void putc(int ch) {
        putchar(ch);
    }
}
#include "IOTest.hpp"
```

注意 _IOFBF，_IONBF 这些东西在 NOI 系列赛事不能使用，不过它们都是常数，可以预先取得它们的值。

结果很明显，显式 fread/fwrite 速度最快。

若需要输入浮点数，直接调用 strtod，然后用其参数 str_end 重定位。若需要输入输出浮点数，可以使用支持自定义 buffer 的 std::strstream，不过要注意这个类在 C++98 中已被弃用。也可以考虑使用 sprintf。

注意使用 strtod/strtol 时务必一次性读取完毕。

Update：经过单步调试追踪到 strtod 在 glibc 中实现。其具体实现在 _____STRTOF_INTERNAL(/stdlib/strtod_l.c) 中，其实现依赖 GMP，性能。。。。不过 scanf 的调用链为 ___scanf(/stdio-common/scanf.c)→ ___vfscanf_internal(/stdio-common/vfscanf-internal.c)→ ___strtod_internal(/stdlib/strtod_l.c)，所以。。。。

glibc 源代码参见https://sourceware.org/git/?p=glibc.git;a=tree。~~这才是正宗的意大利面~~。

19.2.8 快速乘法取模

当模数的平方超过 long long 的表示范围时，可以使用类似快速幂的方式计算快速乘法。

还有一个无法严格证明正确性的 trick：将 $a*b\%mod$ 表示为 $a*b-\lfloor a/mod*b\rfloor*mod$，其中 $floor$ 内部使用 long double 计算。$floor$ 操作直接使用强制转型，因为 $a/mod*b$ 就在 long long 的表示范围内。由于最终结果在范围内，两个乘法事实上计算的是模 2^{64} 意义下的乘法，暂时溢出并没有关系。

实现代码：

```
typedef long long Int64;
Int64 mulm(Int64 a, Int64 b) {
    Int64 res =
        (a * b -
         static_cast<Int64>(
             static_cast<long double>(a) / mod * b) *
             mod) %
        mod;
    return res < 0 ? res + mod : res;
}
```

无法证明正确性的原因在于 long double 的精度是否足够，而且在 MSVC 中 long double 只有 64 位精度而不是 80 位。不过让人放心的是无论是 long double 还是 double，在 $\geq 10^8$ 次随机测试中没有出现错误。

Chapter 20

总结

20.1 基于输入 564
20.1.1 单变量 564
20.1.2 元素序列 564
20.1.3 时间/区间操作序列 565
20.1.4 树 565
20.1.5 图 565
20.1.6 字符串 565
20.2 基于关键字/特征 566
20.3 时间轴有关问题 566
20.3.1 静态问题 566
20.3.2 动态问题 567
20.3.3 时间分治技巧 567
20.4 树上连通块问题 567
20.4.1 无色连通块 568
20.4.2 黑白连通块 568

20.1 基于输入

20.1.1 单变量

- 矩阵快速幂
- BM 算法 + 线性递推
- 生成函数 + 卷积/分治 NTT

20.1.2 元素序列

20.1.2.1 有序序列

考虑相邻元素的关系。

20.1.2.2 无序序列

- 贪心预排序
- 归类一起转移

20.1.3 时间/区间操作序列

- 添加:二进制分组
- 添加/删除:扫描线/时间线段树
- 初始化/修改:CDQ 分治/整体二分/线段树

20.1.4 树

20.1.4.1 无根树

- 统计以某个点为根的信息:LCT
- 统计所有路径信息:点分治/树形 DP

20.1.4.2 有根树

- 静态统计以某个点为根的子树信息:DsuOnTree
- 动态统计以某个点为根的子树信息:动态 DP/树链剖分后转序列问题
- 统计与深度有关的问题:长链剖分

20.1.5 图

- DAG:拓扑排序
- 有向图:SCC 缩点
- 无向图:(广义)圆方树

20.1.6 字符串

- 满足条件的最长子串:二分 +(Hash+ 插桩/SA+ 分组)
- 与 LCP 长度有关:SA
- 与子串个数有关:SAM(动态时使用 LCT)
- 与回文串有关:Manacher、PAM、后缀系列。
- 与前缀匹配有关:KMP/Z algorithm
- 与模式匹配有关:卷积

20.2 基于关键字/特征

20.2.0.1 恰好

- 统计:将恰好改为至少/至多,然后使用容斥解决。
- 最优化:WQS 二分

20.2.0.2 约束

- 布尔条件约束:2-SAT
- 不等式条件约束:
  - 线性规划
  - 网络流
  - 差分约束
- 等式条件约束:拉格朗日乘子法

20.2.0.3 位运算

- 按位贪心
- 线性基
- 子集卷积

20.2.0.4 在指定范围内存在

- 时间线段树
- 扫描线

20.2.0.5 两项乘积小于定值

分块/对数据分治,步长小的一起转移,步长大的单独转移。

20.2.0.6 最大化最小值/第 k 大

使用(整体)二分。

20.3 时间轴有关问题

20.3.1 静态问题

- 对固定区间的结果有影响:将存在转化为扫描线插入 + 删除,变为动态问题。
- 可持久化操作:所有版本形成了一棵树,可以使用 DFS 序将操作展开为序列。

20.3.2 动态问题

在询问中带有修改操作, 修改对后续的询问有影响, 修改之间对询问的贡献独立。

- 仅插入:
  - 离线: 对于后半部分的询问, 前半部分的所有修改均对它们有影响。使用 lg 的代价进行 CDQ 分治。
  - 强制在线: 二进制分组
- 离线 + 插入 + 删除
  - 将其转化为在固定区间内存在, 变为静态问题, 使用时间线段树解决。
  - 对于后半部分的询问, 前半部分在该询问前未被删除的元素有贡献。考虑从右到左扫描后半部分的询问, 使用时光倒流将前半部分的删除转化为插入。再套用仅插入的做法, 以 $\lg^2$ 的代价除去动态插入删除操作。
- 操作将区间内的数变为同一个数

  下面介绍将此类操作改为添加与撤销操作的方法。

  建立一棵以位置为下标的线段树, 将 $[l,r]$ 的元素修改时, 向 $[l,r]$ 处插入时间 + 新元素标记, 若当前节点已有标记则下推标记并清除, 当递归到整层时取出子树内的所有标记并清除。

  记当前时间为 j, 对于取出的时间为 i 的标记, 将其转化为两个操作:
  - 在时间 i 处执行覆盖操作
  - 在时间 j 处撤销操作

  构建时间复杂度 $O(m\lg^2 n)$, 实测时间复杂度接近 $O(m\lg n)$。

  有了撤销操作后, 覆盖操作可以转化为区间加减与区间和查询解决。

20.3.3 时间分治技巧

- 分治前预排序, 分治到子区间时线性划分
- 分治后尽可能线性合并子区间的信息
- 分治计算一边对另一边的贡献时, 考虑不建立数据结构, 使用双指针法线性计算

该内容参考了国家集训队 2013 论文集许浩然的《浅谈数据结构题的几个非经典解法》与国家集训队 2014 论文集中徐寅展的《线段树在一类分治问题上的应用》。

20.4 树上连通块问题

下文参考了 IOI2018 中国国家候选队论文集任轩笛的《解决树上连通块问题的一些技巧和工具》。

20.4.1 无色连通块

20.4.1.1 树形 DP

以 dp_u 表示点 u 的子树内以 u 为根的连通块的答案。
时间复杂度取决于合并儿子的复杂度。

- 若合并复杂度为 $O(1)$, 时间复杂度为 $O(n)$
- 若合并复杂度为 $O(siz_a * siz_b)$, 时间复杂度为 $O(n^2)$

20.4.1.2 DFS 序单点合并

有时子树合并的复杂度过高, 而单点合并较为廉价。

假定连通块必须包含根节点, 以根节点生成 DFS 序, 那么一个连通块就是 DFS 序列上去掉若干段。

记 dp_i 为考虑 DFS 序上第 i 个位置以前的答案。考虑是否选择位置 i, 若选择则从 dp_i 转移至 dp_{i+1}, 否则转移至 $dp_{R[i]+1}$, 表示跳过整个子树, 其中 $R[i]$ 表示以 i 为根的子树区间右端点。

以背包问题为例, 若使用树形 DP 则需要 $O(nm^2)$ 的复杂度, 其中 m 为背包大小。然而 DFS 序单点合并添加物品只需 $O(m)$, 时间复杂度 $O(nm)$。

一般题目不要求连通块必须包含根节点, 那么可以考虑点分治, 对每个重心管辖的连通块做一遍 DP, 时间复杂度 $O(nm \lg n)$。

20.4.1.3 点数-边数

利用树的 "点数-边数 $=1$" 的性质, 枚举计算包含每个点/每条边的方案数, 然后将点的答案减去边的答案, 可以使得所有合法连通块都恰好被计数一次。

这个技巧常用于求若干个连通块的交的场合。

20.4.1.4 线段树合并 DP

若 DP 数组大小不超过子树大小(比如颜色), 且合并时仅仅简单地将对应位合并(位与位独立), 可以使用线段树合并进行 DP 转移, 时间复杂度 $O(n \lg n)$。

20.4.1.5 链分治动态 DP

- 树链剖分 + 矩阵乘法 + 线段树
- 全局平衡二叉树
- LCT

20.4.2 黑白连通块

给黑白颜色各开一个有根 LCT 维护连通性, 然后若该节点为黑色则其在黑色 LCT 中向父亲(给根节点一个虚父亲)连边, 白色亦然。查询连通块时, 在所处 LCT 中找到根, 根是连通块最浅节点的异色父亲, 连通块的信息存储在 splay 链头右儿子中(注意不能是整链信息-链头信息, 考虑白连通块-黑父亲- 白连通块的情况)。

20.4.2.1 多色连通块处理

给每种颜色建一个数据结构维护, 以自身颜色为黑, 其它颜色为白。时间复杂度多一个颜色数的因子。

Appendix A

资料推荐

A.1 个人博客

- Miskcoo http://blog.miskcoo.com/ FFT 与数论
- hzwer http://hzwer.com/ 大量题解
- skywalkert https://blog.csdn.net/skywalkert 数论
- Candy? https://www.cnblogs.com/candy99
- 租酥雨 http://www.cnblogs.com/zhoushuyu/
- VFleaKing http://vfleaking.blog.uoj.ac/
- Picks http://picks.logdown.com/ FFT
- MoebiusMeow https://www.cnblogs.com/meowww
- ACMaker https://blog.csdn.net/ACMaker 旋转卡壳
- ACdreamer https://blog.csdn.net/ACdreamers
- 小蒟蒻 yyb https://www.cnblogs.com/cjyyb/
- fjzzq2002 https://www.cnblogs.com/zzqsblog
- FlashHu http://www.cnblogs.com/flashhu/
- Claris https://www.cnblogs.com/clrs97/
- negiizhao http://negiizhao.blog.uoj.ac/ 毒瘤

A.2 好用的网站/工具

- Wikipedia-EN https://en.wikipedia.org 比百度不知道高到哪里去了
- cppreference https://en.cppreference.com/w/ 查询 C++ 知识必备
- OEIS http://oeis.org/ 整数数列题找规律必备

- WolframAlpha http://www.wolframalpha.com/ 高级计算器
- GeoGebra https://www.geogebra.org/ 优秀的数学软件
- 3Blue1Brown http://www.3blue1brown.com/ 脑洞有趣易懂的数学教程
- Graphviz http://www.graphviz.org/ 图形可视化工具
- Project Euler https://projecteuler.net/ 数学题集
- BZOJ 离线题库(BZOJCH)[By 阮行止] http://ruanx.pw/bzojch/index.html
- BZOJ 题号查找器(BZOJNO)[By 阮行止] http://ruanx.pw/bzojch/bzojno.html
- 马同学高等数学 https://www.matongxue.com/
- 某 OJ https://csacademy.com
- BZOJ 题目一句话题解整理 By CreationAugust
  https://blog.csdn.net/creationaugust/article/details/51387623

A.3 优秀资料

- CodeForces 优秀文章 http://codeforces.com/blog/entry/57282
- E-Maxx 算法合集英文版 https://cp-algorithms.com/
- OI-Wiki https://github.com/24OI/OI-wiki/
- 知乎上 OI 相关文章 https://www.zhihu.com/collection/213577780
- 演算法筆記 江任捷http://www.csie.ntnu.edu.tw/~u91029/index.html
- 一份不太简短的 LaTeX2$_\varepsilon$ 介绍 https://github.com/CTeX-org/lshort-cn/

A.4 推荐书籍

- 《算法导论(原书第 3 版)》ISBN: 9787111407010
- 《线性代数及其应用 (原书第 5 版)》ISBN: 9787111602576
- 《C++ Primer Plus(第 6 版) 中文版》ISBN:9787115279460
- 《Effective C++: 改善程序与设计的 55 个具体做法 (第 3 版中文版)》 ISBN: 9787121123320
- 《More Effective C++: 35 个改善编程与设计的有效方法 (中文版)》 ISBN: 9787121125706
- 《Effective Modern C++(中文版)》ISBN: 9787519817749 (~~依然记得当时在英语作文中引用了这本书并使用书名号~~, ~~被英语老师怼了一通~~。)

这些就是我学习时读过的书, 读书太少姿势不够。~~PBRT3 就不推荐了~~。

A.5 其它

- THUSC2018wys 发言稿 https://wys.life/2018-06-THUSC-speech.html
- 《中国面壁者》中国青年报
  http://zqb.cyol.com/html/2018-04/17/nw.D110000zgqnb_20180417_1-04.htm
- 膜拜 kczno1 和 wxh019010, 代码常数超小
- 自由不等于无拘无束, 不等于不需要负责任。相反, 一些合理的约束可以帮助我们更好地行使自由。

Appendix B

Index

*Constant
  $\gamma \approx 0.5772156649\ldots$, 286
  De Bruijn Sequences 常数 **0x077CB531U**, 134
  NTT 模数 P={469762049, 998244353,1004535809, 2281701377},g=3, 220
  线性同余随机数生成器 $a = 48271, c = 0, m = 2147483647$, 95
*TODO
  2-SAT 构造方案正确性证明, 415
  Alpha-Beta 分支规模大小, 11
  Baillie-PSW 素性测试正确性证明, 146
  Catalan 应用证明, 203
  Dinic with LCT, 29
  FWT 公式推导与记忆, 221
  Hopcroft–Karp 算法, 14
  K-D Tree 查询复杂度证明, 117
  Kernelization, 548
  KKT 条件, 517
  Link Cut Memphis, 568
  Meissel Lehmer, 166
  PAM 统计以下标 i 结尾的回文串数, 452
  Pohlig-Hellman 算法正确性证明, 172
  PollardRho 时间复杂度证明, 146
  Raney 引理证明, 213
  SAMRight 集合性质证明, 460
  Tarjan 算法正确性证明, 413
  zkw 费用流正确性证明, 45
  一般图支配树, 436
  主定理证明, 528
  仙人掌典型应用, 420
  修复三维快速凸包模板, 495
  剩余系列证明与代码, 183
  匈牙利算法标准描述与正确性证明, 14
  区间众数修改操作, 540
  单模数多询问离散对数, 169
  单点度限制最小生成树实现, 402
  单纯形算法正确性证明, 521
  卡 SPFA, 378
  吉普车问题证明, 287
  启发式模拟退火, 513
  完全散列, 128
  并查集复杂度证明, 111
  弦图相关性质证明, 430
  扩展欧拉定理在矩阵幂中的应用, 139
  拉格朗日定理证明, 191
  提高对 Segment Tree Beats 的理解, 69
  无向图最大权匹配, 436
  最值反演推广证明, 285
  最大势算法正确性证明, 426
  最大匹配随机算法实现, 436
  最大权闭合子图算法的正确性, 51
  浮点数编码, 530
  混合基 FFT 的非递归形式, 242
  特征方程相关证明, 293
  特殊图下 Dinic 的时间复杂度证明, 21
  王室联邦分块法的块大小, 541
  直线相交正确性证明, 481
  研究 zkw 线段树课件的错误并给出 RMQ 模板, 63
  约数个数函数前缀和, 166
  线性构造后缀树, 452
  编码字符代价不等的 Huffman, 447
  置换的群性质结合数论的应用。, 199
  补充博弈论经典模型, 11
  解释 KM 算法优化的合理性, 17
  证明 Burnside 引理, 191

证明无向图最小割算法的正确性并修改模板, 42
贝尔三角形的意义, 208
雅礼集训 2018Day8B 的乱搞做法证明, 59
验证 BM 算法的最优性质, 290
验证解释 Shamos 算法, 509

A

Aho–Corasick Algorithm, 442
Alpha-Beta Search, 11
Anti Nim, 6
Arithmetic Function, 150

B

Baby Step Giant Step, 166
Barzilai-Borwein Method, 514
Bash Nim, 5
Berlekamp–Massey Algorithm, 288
Biconnected Component, 414
Binary Search Tree, 93
Bipartite Graph, 13
Birthday Trick, 146
Bland's Rule, 521
Blossom Algorithm, 430
Borvka's Algorithm, 399
Brian Kernigan's Bit Counting, 559
Burnside's Lemma, 191
Bézout's Theorem, 132

C

Cantor Expansion, 212
Catalan Numbers, 201
Cayley's Formula, 405
Chinese Remainder Theorem, 149
Chord, 425
Chordal Graph, 425
Christofides Algorithm, 424
Cipolla's Algorithm, 181
Clique, 424
Completely Multiplicative Function, 150
Constant Function, 151
Convex Hull, 485
Cumulative Distribution Function, 274

D

Dancing Links X, 124
De Bruijn Sequences, 134, 559
De Morgan's Laws, 190
Dinic, 20
Dinkelbach Method, 518
Dirichlet Convolution, 153
Dirichlet Inverse, 154
Discrete Fourier Transform, 216
Discrete Logarithm, 166
Disjoint Set Union, 111
DominatorTree, 436
Dual Graph, 48

E

Euler Totient Function, 150
Euler Tour Tree, 130
Euler's Criterion, 181
Euler's Theorem, 138
Eulerian Path, 421

F

Fast Fourier Transformation, 215
Fast Mobius Transformation, 224
Fast Walsh-Hadamard Transform, 221
Fermat's Little Theorem, 137
FHQTreap, 93
Fibonacci Nim, 8
Finger Search, 97
Fractional Programming, 517

G

Garsia–Wachs Algorithm, 336
Generating Function, 276
Genetic Algorithm, 514
Gomory–Hu Tree, 54
Gradient Descent, 514
Graham Scan, 485
Gusfield Algorithm, 54

H

Half-Plane Intersection, 505
Hall's Marriage Theorem, 18
Hamiltonian Path, 424
Hierholzer's Algorithm, 421
Highest-label push–relabel algorithm, 31
Hill Climbing, 513
Hook Length Formula, 336
Hopcroft–Karp Algorithm, 14
Hungarian Algorithm, 14

I

Impartial Combinatorial Games, 1
Improved Shortest Augment Path, 29
Included Subgraph, 425
Inclusion–exclusion Principle, 190
Interval Graph, 430

J

Jacobi Symbol, 141
Jensen's Inequality, 274

K

K-D Tree, 115
Karush–Kuhn–Tucker Conditions, 517
Kirchhoff's Matrix Tree Theorem, 405
Knuth–Morris–Pratt Algorithm, 448
Kuhn-Munkras Algorithm, 16
König's theorem, 17

L

Baillie–PSW Primality Test, 140
Lagrange Inversion Theorem, 285
Lagrange Multiplier, 515
Lagrange's Theorem, 191
Landau's Theorem, 436
Least Squares, 265
Leftist Tree, 118
Legendre Symbol, 180
Lindström–Gessel
–Viennot Lemma, 438
Linear Congruential
Generator, 95
Linear Difference Equation, 266
Linear Programming, 518
Link-Cut Tree, 99
Longest Common Prefix, 454
Lucas Sequence, 141
Lucas's Theorem, 210

M

Maclaurin Series, 272
Manacher's Algorithm, 448
Max-flow min-cut theorem, 38
Maximal Clique, 424
Maximum Clique, 425
MCMF, 42
Meet In The Middle, 542
Meissel–Mertens Constant, 286
Micali-Vazirani Algorithm, 434
Miller–Rabin Primality Test, 139
Min-Max Search, 9
Minimum Coloring, 425
Minimum Covering Circle, 502
Multiplicative Function, 150
Möbius function, 150
Möbius Inversion, 154

N

N-Position, 2
Nim, 5
NimK, 5

P

P-Position, 2
Package-Merge Algorithm, 445
Palindromic Tree, 450
Perfect Elimination Ordering, 425
Permutation Groups, 191
Pick's Theorem, 484
Planar Graph, 48, 437
Pohlig–Hellman Algorithm, 172
Pollard Rho, 146
Pollard's rho Algorithm
for Logarithms, 169
Primitive Root, 176
Probability Density Function, 274
Purfer Sequence, 350
Pólya Enumeration Theorem, 192

Q

Quadratic Reciprocity Law, 180
Quadratic Residue, 181
Quick Hull, 490

R

Raney Lemma, 213
Rotating Calipers, 508

S

Segment Tree Beats, 63
Shamos's Algorithm, 508
Simplex Algorithm, 519
Simplicial Vertex, 425
Simulated Annealing, 513
Splay, 97
SprageGrundy Function, 2
SprageGrundy Theorem, 3
Staircase Nim, 6
Steiner Tree, 407
Stirling Number, 206
Stochastic Hill Climbing, 513

Stoer-Wagner Algorithm, 38
Strong Lucas Probable
Prime Test, 141
Strongly Connected
Component, 412
Suffix Array, 453
Suffix Automaton, 458
Suffix Cactus, 457
Suffix Tree, 452

T

Taylor Series, 272
Tonelli–Shanks Algorithm, 183
Totally Unimodular Matrix, 526
Touchard's Congruence, 208
Tournament, 436
Travelling Salesman Problem, 424
Twentyfold Way, 200

U

Ukkonen's Algorithm, 452
Unit Function, 151

W

Wythoff's Game, 7

Y

Young Tableau, 336

Z

Z Algorithm, 470
zkw's Segment Tree, 63

Appendix C

Bibliography

[1] Even, Shimon; Tarjan, R. Endre (1975). "Network Flow and Testing Graph Connectivity". SIAM Journal on Computing

[2] Tarjan, R. E. (1983). Data structures and network algorithms.

[3] 胡伯涛《最小割模型在信息学竞赛中的应用》

[4] Thomas H.Cormen / Charles E.Leiserson / Ronald L.Rivest / Clifford Stein. Introduction to Algorithms Third Edition

[5] n+e《K-D Tree 在信息学竞赛中的应用》

[6] M. J. Golin and G. Rote. A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs

[7] L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes

[8] 陈丹琦《弦图与区间图》

[9] S. Micali and V. V. Vazirani, "An O($\sqrt{|v|}|E|$) algoithm for finding maximum matching in general graphs", 21st Annual Symposium on Foundations of Computer Science (sfcs 1980)(SFCS), Syracuse, NY, USA USA, , pp. 17-27. doi:10.1109/SFCS.1980.12

[10] Anant Jindal and Gazal Kochar and Manjish Pal, "Maximum Matchings via Glauber Dynamics",CoRR

[11] Yang Zhe《SPOJ375 QTREE 解法的一些研究》

[12] Robert Baillie; Samuel S. Wagstaff, Jr. (October 1980). "Lucas Pseudoprimes" (PDF). Mathematics of Computation. 35 (152): 1391–1417.

[13] David C. Lay , Steven R. Lay , Judi J. McDonald . Linear Algebra and Its Applications 5th Edition

后记

C.1 初稿

2018 年 10 月 23 日

写这本笔记大概花了我两个月的时间, 此时这本笔记大概有 200 千字 (50 千字为中文), 我为自己的成果感到骄傲。在此期间我系统地复习了 NOIP 复赛以上的知识点, 学到了许多新的方法/技巧/ 思路, 重新理解了之前看不懂的算法, 收获颇多。我将自己的想法记录于此, 以便之后重新翻阅。该笔记在内容上还是较为完整的, 在接下来的时间内我会在巩固的过程中继续学习并补充新的高级算法 /数据结构, 以应战省选和 NOI2019。

在我作为 D 类队员参加了 NOI2018 现场赛后, 我发现了自己的严重问题—---知识点掌握不扎实, 导致我没能拿到该得的分数。所以我决定自己写一写复习笔记来建立出一套自己的思维体系。

同时作为目前校内在 OI 道路上走的最远的人(其他人初赛都过不了。。。孤单), 我没有信息学强校学生所拥有的资源——学长。有学长的指导和遗留下的资料, 能够少走许多弯路。既然自己没有学长, 只好自己做学长了。这本笔记算是是我的一份心意吧, 希望学校的 OI 事业能够有所发展, 学弟们(??? 我好像根本不知道有学弟)如果需要我帮助的地方, 随时联系我。我的表达能力不太好, 而且有些简单的地方没有细讲, 你们可以打开页底下的参考资料链接细看(这些资料是我自己能够接受并理解的), 虽然这些东西在百度和维基上都能轻易找到, 但我之前根本就不知道这些东西的存在(直到考完后才知道, 已经晚了)。。。所以你们就把它当做一种索引吧。

在写这本笔记的过程中, 我发现自己在使用 LaTeX 和排版方面还存在许多问题:

- 目录太长, 分类过细
- 参考资料链接过多
- 不恰当的知识点分类
- 过长的 TODO List
- 滥用 lstlisting
- 书籍排版密度过低
- 滥用 Index
- 未熟练使用 BibTex
- 数学公式排版掌握不熟
- 缺少图片
- 语言表达不当或过于口语化(才发现 ‘‘即可’’ 之类的词是从其它 blog 上学来的)

上述问题将会在接下来几轮的 review 中得到改善。

在此我要感谢学校领导、老师和同学的理解与支持，感谢父母（还有我妹）对我的大力支持。感谢写出优秀博客的 OIers 和 Wikipedia-EN 的维护者，他们为我提供了大量的参考资料与经验总结，使我受益匪浅，我所做的不过是将它们聚集在一块而已。~~差点忘了感谢 CCF~~。

刚开始写这本笔记时，我遇到了一件让我感到最幸福的事：我喜欢的女生向我表白了。在此之前我已经预先认真考虑了几个月，所以当时我毫不犹豫就接受了。那个月是我最幸福的日子，我开始改变自己，更加努力地学习，以应对将来需要承担的责任。她给了我写这本笔记的动力，她能够理解许多别人所不能理解我的事情。我当时认为自己是这个世上最幸运的人。但好景不长，在本笔记初稿即将完成时，我最担心的事情还是发生了——双方突然无话可说。我是预先知道这个问题会发生，而且已经预先提醒过她了——遇到问题要多沟通，商量解决方案。但在短短几天内，她对我的态度越来越差。最后她以一堆自相矛盾的理由离开了我，根本不给我与之沟通的机会。从前的誓言，以及我对她的好，她似乎已经完全忘记了。我不知道她的真实意图，她或许是为了让我安心学习。我仍然相信自己的单元测试是可靠的，她最终会回来的。我决定履行自己之前的诺言：我等她两年。我好想在她身边给她讲解数学题；好想在她生理期时给她端杯热水，陪在她身边；好想和她一起去公园散步，在绿道慢跑，一起看动画电影；好想在她心情不好时，自己在旁边给她疏导，给她一个拥抱；好想静静地坐在一旁，听她弹琴唱歌；好想再让她笑着敲我的头；好想高中毕业后用自己的 Renderer 给她渲一个鸽了两年的生日礼物。唉，这些愿望怕是无法实现了。现在自己还是先努力学习，认真搞 OI，考上理想的大学。~~得之我幸，不得我命。未来会怎么样，得看自己的造化子~~。**I can do it!**

$\frac{\sin 4\theta}{\sin \theta}$，我等你回来。

C.2 二轮复习

2019 年 1 月 3 日

不得不吐槽 Review 花的时间居然比编写还要久。。。

二轮复习后我对该笔记做了大量修改，主要内容如下：

- 补充了大量高级内容
- 补充了刷题过程中的一些解题方法、坑点
- 修改了一些不恰当、不详细的文字表述
- 更正文中错误
- 规范符号与术语的表达
- 解决并同时新增了一些 TODO
- ~~消灭了“即可”~~

NOIP2018 已经过去了，我初赛成绩 86，复赛成绩 496，成绩还算令人满意（这里的人指老师、校领导和局领导们）。虽然我能凭这个成绩去 WC2019 和 THUWC2019，但是我知道自己的实力仍然不够。NOIP2018 暴露了我的另外一个问题：粗心。初赛因两题水题而不能 AK，复赛因没有从理论上论证算法正确性而留下成绩不能上 500 的遗憾（感谢 CCF 负责任的数据）。根本原因仍然是我对知识点的掌握不够熟练。

我接下来的规划：

- 针对薄弱专题继续复习与刷题。

- 巩固新学的知识。
- 在 WC2019 后继续学习新知识。
- 参加线上比赛以提高应试能力和分析思考能力
- 学会使用 Vim, 提高码速

在此我也总结一下我的 2018 吧。

- 2018 年, 我结识了一些 OIer, 见识了各类型比赛, 意识到自己的水平很菜。
- 2018 年, 我借助 CUDA 写了一个玩具级光栅化渲染器/基于物理的渲染器, 感受到了 CG 的魅力, 以及其幕后需要的大量数学/物理知识储备。
- 2018 年, 我经历了从年段第 2 一路跌到年段第九十几名, 再一路从 60,30,15 回到年段第 2 的 ‘‘过山车’’ 式的惊险。在此感谢我的老师对我的鼓励, 段长对我的教导, 以及她在那段时间对我的支持与鼓励。
- 2018 年, 我经历了从喜欢到恋爱, 再到失恋的标准早恋结局(事实上我的遭遇比早恋要惨得多)。我仍然等着她, 因为我对她有承诺, 对自己也有承诺。
- 2018 年, 我发现自己的计算机/OI/数学知识可以应用到许多方面, 包括但不限于:
  - 物理探究课上给使用传感器的物理实验设备安装驱动, 然后被物理老师当做苦力给每台机子都安装一遍
  - 给班里的英语剧做音频剪辑
  - 被地理老师请去给机器人编程
  - 用扒注册表的手段清除了班班通的病毒(顺便安利了一波火绒杀毒)
  - 给数学老师和同学们安利了一波 GeoGebra(垃圾几何画板)
  - 在语文课上放映用 LaTeX 做的 ppt beamer, 惊艳全场
  - 在英语课上高举《线性代数及其应用》给老师讲 ‘‘列’’ 的单词应该是 column
- 到了年底, 感觉到自己的身体快撑不住了(或许是因失恋而过度伤心)。于是我提早了运动计划的实施(本来想在退役后才开始的), 坚持每天晨跑。一个星期后, 困扰我多年的过敏性鼻炎有了好转。至此以后我上课更加精神了, 因此被英语老师表扬。上个星期我还被路上的一位老人表扬了, 继续坚持!!!

没有她的这段时间里, 起初我是十分痛苦的。我的人生计划第二次被她打断, 原来幻想的幸福美满的家庭成了泡影。那段时间我一直都在听 Right Here Waiting。后来我意识到再这样颓废下去, 不仅这个梦想破灭了, 另一个梦想也将受到影响。如果这两个梦想能够实现一个, 我就很满足了。偶然有一次我听到《中国男儿》后, 更加坚定了改变自己的决心。我第一次听这首歌是在电视剧《五星红旗迎风飘扬》中。片尾氢弹爆炸的场景, 以及这首歌的旋律和歌词, 给我的印象很深。

歌词一起放在这里:

中国男儿, 中国男儿, 要将只手撑天空。
睡狮千年, 睡狮千年, 一夫振臂万夫雄。
长江大河, 亚洲之东, 峨峨昆仑, 翼翼长城,
天府之国, 取多用宏, 黄帝之胄神明种。
风虎云龙, 万国来同, 天之骄子吾纵横。
中国男儿, 中国男儿, 要将只手撑天空。

睡狮千年, 睡狮千年, 一夫振臂万夫雄。
我有宝刀, 慷慨从戎, 击楫中流, 泱泱大风,
决胜疆场, 气贯长虹, 古今多少奇丈夫。
碎首黄尘, 燕然勒功, 至今热血犹殷红。
中国男儿, 中国男儿, 要将只手撑天空。
睡狮千年, 睡狮千年, 一夫振臂万夫雄。
长江大河, 亚洲之东, 峨峨昆仑, 翼翼长城,
天府之国, 取多用宏, 黄帝之胄神明种。
风虎云龙, 万国来同, 天之骄子吾纵横。

前几天我又看了《中国面壁者》, 感触很深。或许这才是我的归宿吧。

她提出分手前, 我正在对她的感情稳定性进行测试(我是个奇怪的~~人~~计算机)。当她说出我们已经无话可聊时, 我的内心是欣慰的。我努力了这么久, 终于构造出了一个极端条件, 这是一个很好的锻炼机会。但是她的下一句 "分吗" 如同晴天霹雳。既然她不喜欢我也不再理我了, 我的测试也变得既没有意义也不能实行。我只能努力改变自己, 做好充分准备, 抓住一切机会。

出了这档子事, 我的职业选择也不再局限于 NVIDIA 那种自由的工作了, 本来想多陪陪家人的。现在我感兴趣的职业方向如下:

- CG: Disney 的动画电影对我影响很深, 所希望自己的技术能够协助电影艺术家创作出优秀的动画电影, 向世人宣传正能量
- HPC: 指挥一堆 CPU 协作和指挥千军万马一样浪漫

一言难尽啊。。。。 适可而止吧, 睡觉时间快到了。
比赛在即, 我要更加努力。
THUWC2019&WC2019&FJOI2019 Round 1 加油!!!

C.3 三轮复习

2019 年 4 月 4 日
一轮更比一轮咕。
这三个月来我对该笔记做了大量修改, 主要内容如下:

- 补充了一些学习计划外的技巧(需~~求刺激进步~~)
- ~~升级了 Checker~~
- ~~将某些连自己都看不懂的文字表述修改为将来的我看不懂的文字表述~~
- 准备对解题思路程序化, 添加了几类问题的系统解题思路。未来可能会做思维导图?

再记一下上一轮复习牵挂的事:

- THUWC2019: 进了面试, 口语测试体验极差(~~老师你刚才只叫我读啊, 我就一个单词一个单词地念啊。老师: 好子你可以出去子。~~), 拿到了神奇的三等约: 再来一次。
- WC2019: 冬眠营果然名不虚传, 我从去广州开始一直睡到了 FJWC2019。考试体验极差, 交互题暴零又让我拿了一次 Cu(要是 CTSC2018 没有 Day3 我就可以达成集齐 NOI 系列赛事 Cu 的成就了, 可惜今年为了不浪费时间没报 CTSC 和 APIO)。考试前还被去年 CTSC/APIO 的室友嘲笑了一番: 你怎么看起来这么落寞啊。在考试前知道了他去年 PKUWC 已经签了无条件本一, 进考场时心里很不是滋味。

- FJOI2019 Round 1:鸽到和 FJOJ2019 Round 2 一起进行,还有一个星期左右。不过从 FJWC2019 的模拟成绩来看,我连 D 类都够不着。
- 她:与我不再有任何联系,不过看她的学习和生活没有受到任何影响我也就放心了。

接下来的计划:

- 完成系统解题思路梳理,把会的东西全部挂上去,不熟的东西舍弃掉。
- 继续按照专题刷题。
- 从头开始 Review(下一次后记的时间会不会咕到退役?)。
- 省选后开始看集训队论文。
- 刷各大赛事的题目。

她离开后我感觉心里空落落的,有开心的事没有人分享,受到了打击也没有人安慰。再加上这一个月多都在家里,每天除了吃饭睡觉散步,其它时间都待在电脑面前,一天没说几句话。我感觉自己快要疯了,白天有时会摸鱼看新闻写 Checker,晚上有时候写题写到一点多(有时在写 Checker),在自己房间里会颓废看视频,会一个人和自己用记事本聊天,会一个人缩在被子里泣(《水浒传》第二十五回 王婆计啜西门庆 淫妇药鸩武大郎:看官听说:原来但凡世上妇人,哭有三样:有泪有声谓之哭,有泪无声谓之泣,无泪有声谓之号。)。我不知道再这样下去自己的生理和心理会不会崩溃,写初稿的第一个月自己可不是这样的。只能强制自己死撑了,崩溃了也算是一种解脱,只是感觉对不起身边对自己寄予厚望的人们。好怀念那一个月的时光啊,那可能是我最幸福也是最痛苦的回忆了。

对于她的行为,我不能理解,无法接受,但是必须尊重。一年后的生日礼物按照计划准备,但自己不想再打扰她,还是自己保存着吧。对于自己的感情,认识她之前我就已经跟自己讲得很清楚,只能按照自己的承诺继续等。虽然复合的希望渺茫(我也不知道即使她愿意复合,我会不会再次信任她),但这种来自直觉的决策应该是正确的,我等她两年。

感觉自己离开课堂好久了,有点厌学,根本不知道老师到底在讲什么,我已经没有退路了。moe 又在今年开始缩减自主招生规模,提高要求,更是把我逼到绝境。要是像室友那样早一年进入面试,或许自己就不会遭受这么多打击了。看着他签了无条件本一后一脸轻松的样子,自己很是羡慕,这两年省排名在自己左右的人基本都签了,就我只有一张废纸。

目前还有两次机会,我希望自己也能有一个 True Ending。

C.4 FCS NOI2019 随笔

C.4.1 day0

看着自己的笔记发现自己学了好多,这几天也不知道自己要做什么了。希望有备而来的今年能比懵逼的去年考得更好。

真的好害怕自己在考场上又想起她。。。只要题目不会做我就会想她,然后崩盘(从 NOI2018 到 WC2019 都有过)。而且现在已经分开了,我再也没办法找她要安慰了。今天晚上再小小地哭一会吧,明早考试应该情绪会好些。

我感觉自己好奇怪,过着和正常学生不一样的生活。每天在不同计算机的面前切换,一整天很少说话,在学校也很少参加集体活动,常年失踪,在同学眼里我就是个呆子。

但我想说,OIer/程序猿不是这样的!!! 他们也可以多才多艺,也可以有丰富的生活,而不是大家公认的“人傻钱多死的快”。只不过,我也不是正常的程序猿,我让身边人对程序猿的认知出现了偏差。我到现在还记得初中毕业班的英语老师说的“不要像那些程序猿一样”

可能这里还有一些心理问题吧。不知道为什么，自托儿所开始就不合群，但是我一直都希望自己能够融入集体。但我努力学习了，成为了老师眼里的乖孩子，但是换来的只不过是同学们的尊敬（仍然记得一年级被段霸放过的场景），而不是关注。为了刷点存在感，我经常用超出课程的数学来给同学们讲题，但是我看到的不是同学对未知的渴望，而是在讲台下各做各的。有时甚至连老师都没听，一个好的 idea 就这样被埋没了，变成了我的个人表演。现在我在想出一个 idea 后，也不大愿意与大家分享了，自己晚上回去哭一哭吧，祭奠一下它。分手前想给她做的题，在分手后 18 天我终于想出了一个绝妙的纯几何方法，可惜连这个唯一的听众也离我而去。

所以我的心理才很脆弱啊。如果外校 OIer 对我防备，对我不理不睬，或者是父母说我这不会那不会，我会哭一晚上。但如果有人对我很好，关心我（比如她），我可以给出一生的承诺。那是我第一次收到生日礼物，那是我第一次与女生有长时间的对话，那是我第一次能够把自己的糗事分享出来，那是我第一次去关心一个人的爱好。这就是喜欢的感觉？但是现在不存在了，那些是第一次，也可能是最后一次。

我实在是不理解，为什么她会把生活过的精彩说得这么容易。在高一，唯一可以调剂我的生活的，只有与她聊天和捣鼓 CG，在高一末被段长劝退 CG 后，她就是我的唯一。我也考虑过，在自己没有自己的爱好后，就会对她特别依赖，这会导致不正常的结果。在她表白前也以影响学习为由询问她要不要暂停接触。可最后还是没有狠下心，造成了现在这个尴尬的局面。我不后悔自己喜欢她，但很疑惑当初为什么她会喜欢我。

有的时候我在想，自己为什么会呆在计算机面前，而不是像正常高中生一样过着精彩的生活。最让人伤心的，莫过于别人在科技文体艺术节、在元旦晚会上唱歌，而我却在机房里刷题。就算我去参与这些活动也做不好，现在的我，离开了计算机，什么都不是。~~给我一台计算机，我什么都能干~~！

可能身边人都觉得我状态很好（没人发现我失恋了），天天乐呵呵的，但他们不知道我在被子里因为一些人的不理睬、防备、嫌弃哭了多久才假装自己很正常。

好羡慕 debug18，能和女朋友一起上 THUhttp://debug18.com/posts/my-2018/。

好羡慕 zhblue，能够引导自己的孩子探寻未知http://www.hustoj.com/?p=1331。

但对于我，她离开后，这些梦可能都无法实现了。但我希望自己的 QQ 头像永远不换。

希望这些东西在 deadline 之前不要被她知道，不希望因为这个而影响她对自己幸福的选择，不过或许这些东西只会让大家觉得我更幼稚罢了。

~~似乎跑题子。。。~~ ~~这是我的书，只要不是政治敏感的东西，爱写啥写啥。好像只有第一行与标题有关~~。

C.4.2 day1

凉凉。。。没有一题会写，沦为暴力选手。退役预定。

上天真的要捉弄我吗？我昨晚还说考试时不要想起她，今天外面就响起了她唱过的歌。。。特大声。外面下着大雨，与我的处境差不多吧。

机会越来越少了，希望越来越渺茫。继续踏实干吧。

C.4.3 day1 下午

居然只有 5 分。。。NOIP 那一点优势完全不存在了。省选就像玩一样。退役已定，可以准备好迎接惨淡的人生了。

她。。。复合的概率只剩下 bias 了吧。

晚上不知又要哭多久。。。

人生没有 Debug 啊，不能断点，不能单步，像是并发。

我不知道自己还能撑多久，在被宣判死刑前，把自己一直以来的梦说出来吧。

- 有一个稳定和谐的家庭
- 捣鼓 CG

- 有空时在县里开班招生, 造学弟
- 吃吃吃

全凉了。
晚上学点 Rust 放松一下, 明天可能没心情考试了吧。

C.4.4 day2

为什么前两题这么水。。。对于这套题目我一点优势都没有。
果然一堆人 200+。
退役。老爸的 Plan B 也该泡汤了罢。
两个月的养老生活结束了, 再也不能待在家人身边。自己的猜想果然不错, 校领导知道没戏后, 就叫我滚回去学文化课了。
混了两年换来这个结果。不仅没得到想要的, 还失去了许多。。。这就是人生?

C.5 End

2019 年 5 月 20 日
最后一次比赛后我就再也碰不了电子设备了, 所以这里提前做个收尾。
THUSC2019 应该是我 OI 生涯中的最后一站吧。我也大概知道了结局, 否则也不会花一个月的时间搞 THUAC, 甚至抽时间刷电影, 这两个是我一直想做但忍着不做的事。
我没能达到自己的要求, 既不能留住她, 也不能让自己走出失恋的阴影, 从而丧失刷题的动力。
我还有一堆的 TODO 没有做完, 如果我能撑到高中毕业而没有崩溃的话, 我会抽时间完成它的。但是不再加入新的东西。如果有学弟愿意接手这个烂摊子, 可以发 Pull Request 给我。
至于开班造学弟的事情, 由于我可能上不了很好的大学, 会成为反面教材, 并且加上近年来 moe 对竞赛方面自主招生的打压, 应该不会有家长送人来, 学校也不会配合的。我能留给学弟的, 只有这本笔记, 附赠一个 Checker。
送给学弟们的忠告:

- 适当地娱乐, 要注意调整比例。我当时靠着鼓捣 CG 还能保持 0 娱乐, 离开它后整个人就变得毫无生气, 离开她后就开始颓废。
- 尽量组团学习, 同等级 OIer 之间的交流可以大量节约时间, 而且不会感到孤独。
- 我在笔记中标记的坑, 一定要记住。这是我用时间与分数换来的。
- 多训练思维。在掌握笔记中的知识后, 你们能做的只有裸题而已。要思考如何将考点隐藏, 如何转化。
- 多出题。上面一条我也做不到, 因为我能接触到的题目少。如果你们是组团学习的, 可以考虑互相出题, 互相 Hack, 这同样是一种训练。多 Hack 可以熟悉出题人可以怎样卡掉某个算法。
- 多打比赛。平时训练和比赛时的心理与策略是十分不同的。缺少比赛也是我的不足。
- 所以说, 只要人数多, 你们就可以自己出题, 举办校赛, 赛后 hack, 这样的训练效果比我自己一个人刷题好得多。
- 同样重视数学和编程方面的训练。虽然 OI 侧重于算法设计, 但是数学功底好就可以快速推导式子, 提高对语言的掌握程度可以减少 bug。

- 如果有事找我, 请确保你的 C++ 代码大括号不换行。其它的代码风格我可以兼容, 但是大括号免谈。

希望将来我校有人能完成我未能完成的梦想, 最好能挑大梁, 培养更多的学弟学妹。
希望这本笔记对你们有一些帮助。

Figure C.1: 这应该是本笔记唯一一张图了